Mapping von Algorithmen und funktionalen Systembeschreibungen auf Architekturen unter Nutzung von Optimierungsstrategien

Entwurf und Implementation einer Java-Applikation

Masterarbeit

zur Erlangung des akademischen Grades Master of Science

vorgelegt der Fakultät für Informatik und Automatisierung

der Technischen Universität Ilmenau

von

Andrey Kondrat

Fachgebiet:	Rechnerarchitektur
Verantwortlicher Professor:	Prof. Dr.-Ing. habil. Wolfgang Fengler
Hochschulbetreuer:	Dipl.-Inf. Marcus Müller
Ort und Datum der Einreichung:	Ilmenau, den 04.05.2010
Inventarisierungsnummer:	2010-002-MA II

URN: nbn:de:gbv:ilm1-2010200207

Kurzfassung

Die Suche nach der optimalen und der schnellsten Lösung eines Problems durch die effektive Benutzung der dem Nutzer zur Verfügung stehenden Ressourcen ist heutzutage eines der wichtigsten Probleme im Bereich der Hochleistungs- und Hochparallelrechnersysteme.

Die vorliegende Arbeit beschäftigt sich mit der Konzeption und Entwicklung einer Java-Applikation für die Transformierung von Algorithmen und funktionalen Systembeschreibungen in ein Datenformat, das von der Architektur des Zielrechnersystems unabhängig ist. Als solches Datenformat wird das Datenflussgraphenmodell ausgewählt, das auf der vom Nutzer vorgegebenen Architektur des Zielrechnersystems der bestimmten Ausführungsstrategie entsprechend partitioniert wird. Die Partitionierung erfolgt durch die Verwendung der qualitativen und quantitativen Graphenanalyseverfahren, mit diesen werden die wichtigen Parameter wie die minimal mögliche Ausführungszeit des Graphen auf dem Rechnersystem mit unbegrenzten Ressourcen und der maximale Speedup berechnet, damit man die minimal erreichbare Ausführungszeit des Graphen auf dem vorgegebenen Rechnersystem bewerten kann.

Die entwickelte Applikation ist nach dem Frameworkkonzept gebaut. Dieses Konzept ermöglicht die Erstellung wiederverwendbarer erfolgreicher Architekturen, Komponenten und Programmiermechanismen und macht die Applikation selbst flexibel und erweiterungsfähig.

Abstract

The searching for the fastest and optimal solution of a problem through the efficient use of the available ressources is at present the most urgent and relevant issue in the field of the high-performance computing (HPC).

This master's thesis presents a concept and an implementation of the Java-application for the conversion of the algorithms and functional system descriptions into a data format that doesn't depend on the architecture of the target computer system. A data-flow graph can be chosen as such a data format and the nodes of it are being mapped to the target computer architecture in compliance with the mapping strategy assigned by the user. The node mapping is carried out by the means of the data-flow graph analysis techniques, which help to compute the parameters of the graph, such as the minimal possible time for the computation of the all graph nodes on the computer system with the unbounded resources and the maximal speed-up, in order to evaluate the minimal execution time of the graph on the computer system established by the user.

The implemented Java-application is based on the application framework concept that enables the developed program components and mechnisms to be repeatedly used and provides for the application itself additional flexibility and expandability.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Im Bereich der Rechentechnik entstehen heutzutage die großen Probleme damit, wie man die vom Nutzer gewünschte, vom Auftraggeber vorgegebene oder minimal mögliche Ausführungszeit des Problems auf dem Zielrechnersystem erreichen und damit die Leistung von diesem optimal ausnutzen kann. Die Hersteller der Hochleistungs- und Hochparallelrechnersysteme bieten voneinander unterschiedliche Mögliglichkeiten, die für die Lösung der verschiedenen Problemtypen geeignet sind, und stellen durch die Hierarchie der Rechnerressourcen (normalerweise Prozessoren), des Speichers und der Kommunikationsressourcen die Mechanismen zur Verfügung, um die Leistung des System zu erhöhen.

Ein weiteres aktuelles und wichtiges Problem in diesem Zusammenhang ist, dass es sehr schwierig ist, ein Modell zu entwickeln, mit dem man die vom Nutzer vorgegebenen Probleme auf jedem beliebigen Rechnersystem modellieren kann, weil sowohl die Probleme in universeller „Sprache“ geschrieben werden müssen als auch in der Modellbeschreibung des Rechnersystems alle strukturellen und funktionalen Eigenschaften von diesem berücksichtigt werden müssen.

Im Rahmen dieser Arbeit soll ein Algorithmus vorgelegt werden, der standardisierte Daten, wie z.B. die mathematischen Formeln, XML-Dateien u.ä., in das universelle Format (Datenflussgraph), das für jedes Rechnersystem „verständlich“ ist, umwandelt und dieses auf das vorgegebene Rechnersystem partitioniert.

1.1. Zielstellung

Wie bereits zuvor erwähnt, gibt es als Voraussetzung zwei Mengen, eine Menge von parallelisierbaren Funktionen und eine Menge von Berechnungsressourcen, und das

Problem ist die zeitlich optimale Abbildung von der ersten Menge auf die Zweite zu finden. Dieses Problem ist NP-vollständig und soll in dieser Arbeit heuristisch durch die Kombination von Graphenanalyseverfahren gelöst werden.

Dafür soll ein Javaprogramm entwickelt werden, das vom Benutzer verwendete Eingangsdaten „versteht“ und diese in übersichtliche und verständliche graphische Form transformiert. Das Programm soll in *Eclipse* als Entwicklungsumgebung realisiert werden und soll auf dem im Fachgebiet Rechnerarchitektur entwickelten Framework und Modellen basieren.

1.2. Vorgehensweise

Das Prinzip, nach dem diese Masterarbeit strukturiert ist, ist der schrittweise Übergang von der groben Beschreibung der fünfstufigen Lösung des oben erwähnten Problems, die im Kapitel 2 erläutert wird, zu der detaillierten Betrachtung jeder Stufe (*top-down* Entwurf).

Im Kapitel 3 werden die theoretischen und funktionalen Grundlagen der Entwicklung von der Javaapplikation, die Struktur von dieser, sowie die Beschreibungen der verwendeten Modelle und die Parameter für die Bewertung ihrer Qualität beleuchtet.

Kapitel 4 widmet sich den Implementierungen von jedem Applikationselement, das sind die Eingangsdatenparser und der Modellkonverter.

Im letzten Kapitel erfolgt eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick auf weiterführende Aufgabenstellungen.

Kapitel 2

Grundlagen

Die Masterarbeit beschäftigt sich mit der Konzeption und Entwicklung einer Javaapplikation zur Partitionierung der verschiedenen Funktionsblöcke auf eine verteilte parallele Plattform. Aus diesem Grund wird in diesem Kapitel sowohl eine Herangehensweise des Entwicklungsprozesses von dem Eingangsdaten- bis zum Zielmodell als auch die Struktur der Architektur des parallelen Zielrechnersystems vorgestellt.

Bei der Beschreibung der Herangehensweise wird vor allem berücksichtigt, welche Entwicklungsstufen durchlaufen werden müssen, um die Eingangdaten ins Format des Zielmodells, d.h. *des Mappings*, zu transformieren. Es wird für jede Stufe gezeigt, welche Datenformate bei diesen angenommen und ausgegeben werden. Weiter in dieser Arbeit werden die Beschreibungen und Softwarerealisierungen von allen diesen Stufen vorgestellt.

Beim parallelen Rechnersystem interessieren, im Rahmen dieser Arbeit, einerseits die Struktur von diesem, d.h. die Anzahl und die Eigenschaften der Prozessoren, sowie die Mechanismen der Kommunikation zwischen diesen.

2.1. Entwicklungsprozess

Den Entwicklungsprozess kann man in die folgenden 5 Stufen teilen, die auch als Blöcke in Abb. 2.1 dargestellt sind:

1) Die Auswahl der Eingangsdatenformate
2) Der Aufbau des Graphenmodells
3) Die Auswahl einer Optimierungs- und Ausführungsstrategie
4) Die Graphenmodelloptimierung und Partitionierung

5) Die Ausführung

Zuerst muss man die für die Untersuchung benötigten Eingangsdatenformate auswählen, diese mit der zu entwickelnden Javaapplikation auslesen und daraus eine graphische Darstellung aufbauen. Im Rahmen dieser Arbeit werden 2 Arten von Eingangsdaten betrachtet, das sind die mathematischen Formeln und die Dateien des MatLab-Simulinkmodells und diese werden in den Graphen gemäß dem Graphenmetamodell transformiert. Die Knoten des Graphen haben für jedes Datenformat eine eigene Bedeutung. Die Umwandlung der Eingangsdaten in einen Graphen wird mit Hilfe des Eingangsdatenparsers durchgeführt. Die Struktur und der Algorithmus des Parsers, sowie die Beschreibung und Spezifikation der Eingangsdaten und des Graphenmetamodells werden im Detail im Kapitel 4 beschrieben.

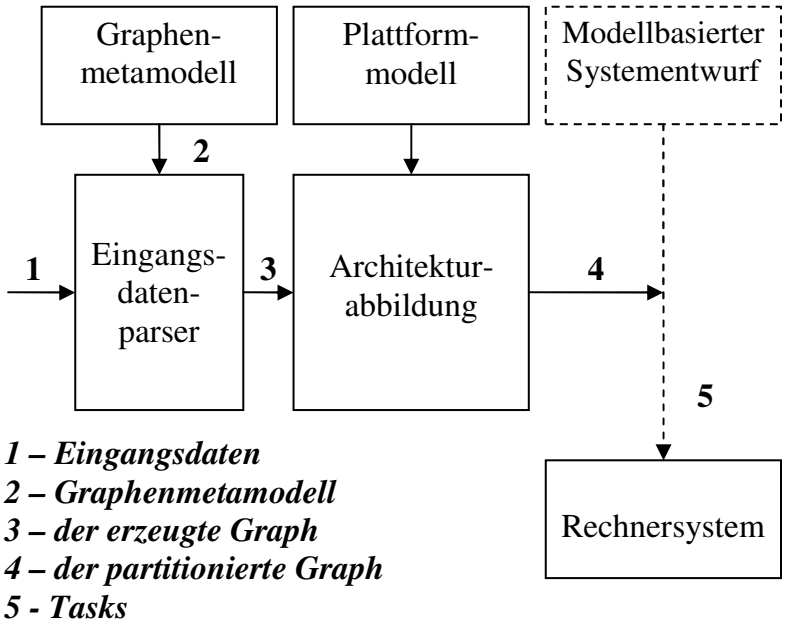


Abb. 2.1. Entwicklungsprozess

Auf der 3. Entwicklungsstufe werden die Optimierungs- und die Ausführungsstrategie ausgewählt, die zusammen eine gesamte Partitionierungsstrategie des Graphen bilden. Zum Schluss wird der Graph der Partitionierungsstrategie und dem Plattformmodell entsprechend auf die Prozessoren und Busse des Rechnersystems partitioniert und dannach als ein Task ausgeführt. Der gesamte Prozess der Partitionierung sowie die dafür benutzten Strategien werden teilweise im Kapitel 3 und detailliert am Ende des Kapitels 4 betrachtet.

2.2. Rechnersystem

Als Zielsystem für die Ausführung der Partitionierung wird das im Fachgebiet Rechnerarchitektur der TU Ilmenau aufgebaute Multi-Digital-Signal-Processor-System (MDSP) benutzt. Die Struktur dieses Systems ist in Abb. 2.2 dargestellt.

Das MDSP ist ein Mehrprozessorsystem mit sechs Prozessoren. Die Prozessoren, die sich auf Modulen befinden, sind über einen Bus verbunden und nach dem Master-Slave-Prinzip organisiert. Das bedeutet, dass einer der Digital Signal Processors (DSP) die Rolle des Masters einnimmt und diesem weitere vier Prozessoren als Slave unterstellt sind. Ein anderer Prozessor übernimmt die Kommunikation mit der Systemumwelt, dieser ist nur mit dem Master verbunden und wird USBMaster genannt. Das System ist so aufgebaut, dass sich jeder Prozessor auf einem Modul befindet. Dieses Modul beherbergt neben dem Prozessor einen eigenen Speicher [1].

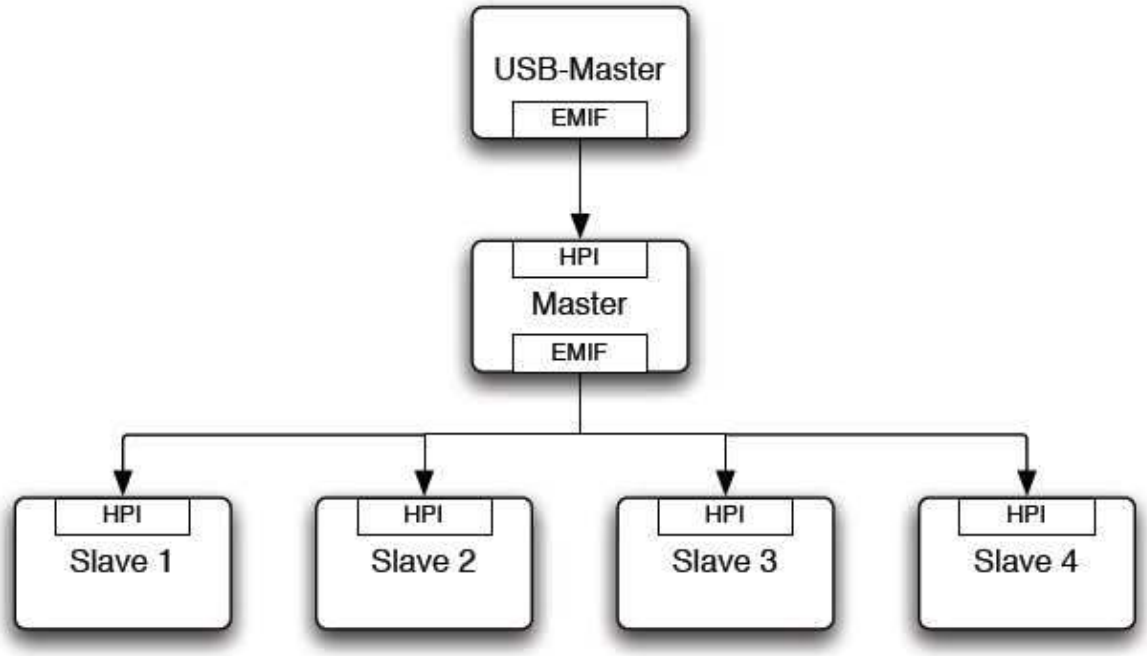


Abb. 2.2. Systemstruktur des MDSP-Systems

Die Kommunikation zwischen den Prozessoren erfolgt unidirektional. Der Master-DSP kann auf die Speicher der Slaves zugreifen. Dazu nutzt der Master eine mit dem External Memory Interface (EMIF) realisierte Speicher-Schnittstelle, die über einen HPI-Bus mit dem jeweiligen Host Port Interface (HPI) der Slaves verbunden ist. Das HPI

erlaubt den Zugriff auf die Speicher der Slave-DSPs. Der Einsatz des EMIFs bietet dem Master die Möglichkeit, über einen Speicherzugriff direkten Zugriff auf den Speicher der Slaves zu erhalten. Welcher DSP-Speicher adressiert wird, wird über den Speicherbereich des Zugriffs ermittelt. Das HPI ist über einen Extended Direct Memory Access-Kanal (EDMA) des EDMA-Controllers mit dem Speicher des jeweiligen Slave verbunden. Dies bedeutet, dass ein Zugriff, bis auf die Arbitrierung, unabhängig vom Slave und ohne diesen zu belasten, erfolgen kann [1].

Die Kommunikation kann nur durch den Master stattfinden, die Slaves verhalten sich passiv. Diese Aufteilung weist dem Master eine reine Kommunikationsaufgabe zu, er soll Daten zwischen den Slave-Modulen vermitteln, d.h. den auf der 4. Stufe des Entwicklungsprozesses erzeugte Partitionierungsalgorithmus realisieren. Die Slaves sind so jeglicher Kommunikationsauflagen entbunden.

Im Rahmen der Applikationsentwicklung wird das Multi-DSP-System als Plattformmodell mit den gleichen Struktureigenschaften dargestellt und damit werden im Kapitel 4 die quantitativen Bewertungen berechnet, wie z.B. den maximalen Speedup, den man für den gegebenen Graphen mit diesem Rechnersystem erreichen kann.

Kapitel 3

Konzept der Applikation

In diesem Kapitel sollen die wichtigsten Konzepte und funktionalen Grundlagen der entwickelten Applikation vorgestellt werden. Einleitend wird *das* für die Erarbeitung der Applikation verwendete *Framework* beschrieben und ein kurzer Überblick auf das Frameworkkonzept als Mittel zur Realisierung wiederverwendbarer objektorientierter Software gegeben. Weiterhin sollen die einzelnen Komponenten dieses Frameworks und die Struktur von diesen detailliert im Rahmen des *Slice*konzeptes betrachtet werden.

Im zweiten Teil dieses Kapitels werden die Anforderungen an den Konverter aus einem *Slice* des Modells zum anderen und die Bewertungscharakteristiken von diesem vorgestellt.

3.1. Frameworkkonzept

Bevor das für die Entwicklung der Applikation verwendete Framework selbst präsentiert werden kann, sind einige theoretische Erläuterungen nötig.

Ein Framework ist selbst noch kein fertiges Programm, sondern stellt den Rahmen, innerhalb dessen der Programmierer eine Anwendung erstellt, zur Verfügung, wobei u. a. durch die in dem Framework verwendeten Entwurfsmuster auch die Struktur der individuellen Anwendung beeinflusst wird [2].

Ein Framework gibt somit in der Regel die Anwendungsarchitektur vor. Dabei findet eine Umkehrung der Steuerung *(Inversion of Control)* statt: der Programmierer registriert konkrete Implementierungen, die dann durch das Framework gesteuert und benutzt werden, statt – wie bei einer Klassenbibliothek – lediglich Klassen und Funktionen zu benutzen.

In dieser Arbeit wird ein *application framework* benutzt. Die Anwendung von *application frameworks* (Applikationsframeworks) eröffnet einen Weg, auf dem es möglich ist, das Wesen erfolgreicher Architekturen, Entwurfsmuster, Komponenten und Programmiermechanismen wiederverwendbar zu machen [3].

Die Grundstruktur eines Applikationsframeworks ist in Abb. 3.1 dargestellt.

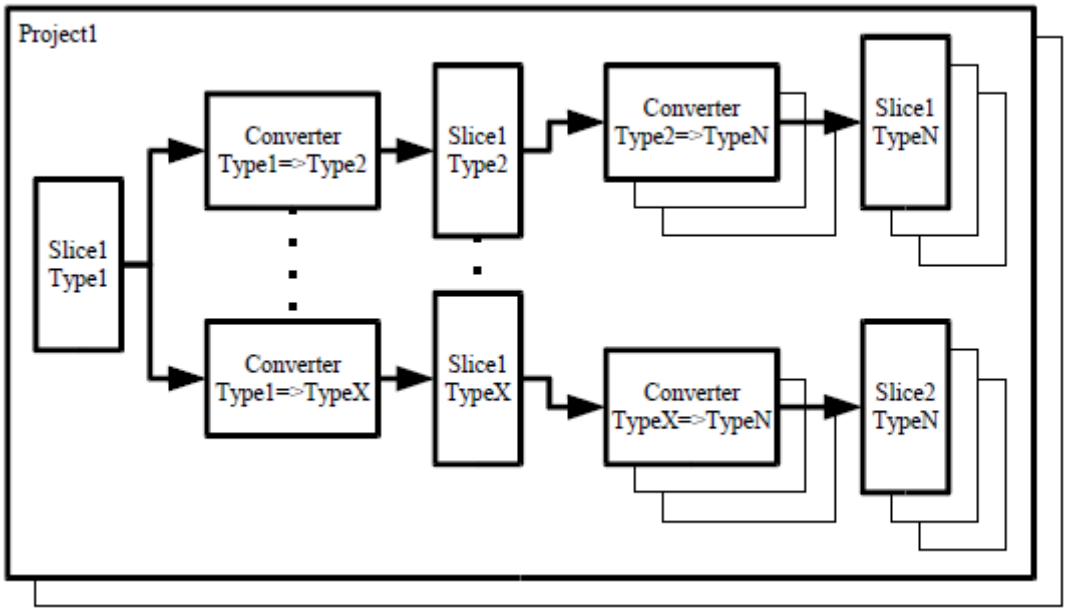


Abb. 3.1. Grundstruktur eines Frameworks

Die umgebende Struktur soll als *Projekt* bezeichnet werden und dient als Container für die Umgebung des Initialmodells und die daraus hervorgehenden Modellumgebungen. In einem Framework können theoretisch mehrere Projekte verwaltet werden. Die Modellumgebung wird wegen ihrer Eigenschaft, einen scheibenartigen Abschnitt der Applikation zu verkörpern, als *Slice* bezeichnet. Jeder *Slice* ist auf einen speziellen Modelltyp zugeschnitten, denn er beherbergt ein Datenmodell von genau diesem Modelltyp und eine Reihe von Komponenten, die nur auf diesem Typ arbeiten können. Die innere Struktur eines Slice wird in Abb. 3.2 näher betrachtet. Die Modellumgebungen sind über Konverter verbunden, von denen jeder die Umwandlung von je einem Modelltyp in einen anderen vornehmen kann [4].

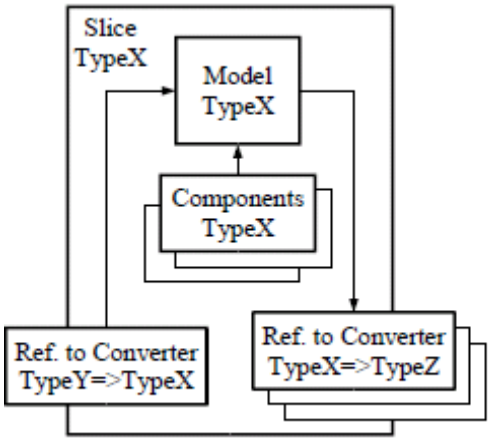


Abb. 3.2. Aufbau einer Modellumgebung (*Slice*)

3.2. Applikationsstruktur

Die Struktur der entwickelten Applikation im Rahmen des dazu verwendeten Frameworks ist in Abb. 3.3 dargestellt. Das Framework selbst ist auch im Detail in [4] beschrieben.

Am obersten Niveau der Applikation befindet sich das gesamte Projekt *DFMapTest*, das aus 2 Slices *ArchmapDFInputSlice* und *ArchmapResultSlice* besteht. Jeder Slice enthält ein grundlegendes Modell *ArchmapModel,* das bei der Konvertierung im 2. Slice erweitert wird, und ein *ModelSaver*, der für die Speicherung der beim Starten der Applikation erzeugten Modellstruktur benutzt wird. *ArchmapDFInputSlice* hat u.a. in seiner Struktur die zwei Importer für die Herunterladung des Modells, das mit der Applikation bearbeitet werden muss.

3.2.1. Modellimporter und Modellsaver

Die beiden Modellimporter, das sind *MatlabMdl2ArchmapDFGraphImporter* und *MathExpr2ArchmapDFGraphImporter*, haben im Prinzip die gleiche Struktur. Zuerst lesen sie die MatLab- bzw. Textdatei ein und dann geben sie diese als String dem jeweiligen Parser weiter. Der Parser analysiert seinerseits diesen Stringdatenfluss und transformiert ihn ins Format des Zielmodells. Die Algorithmen der beiden Parser, die Prinzipien des

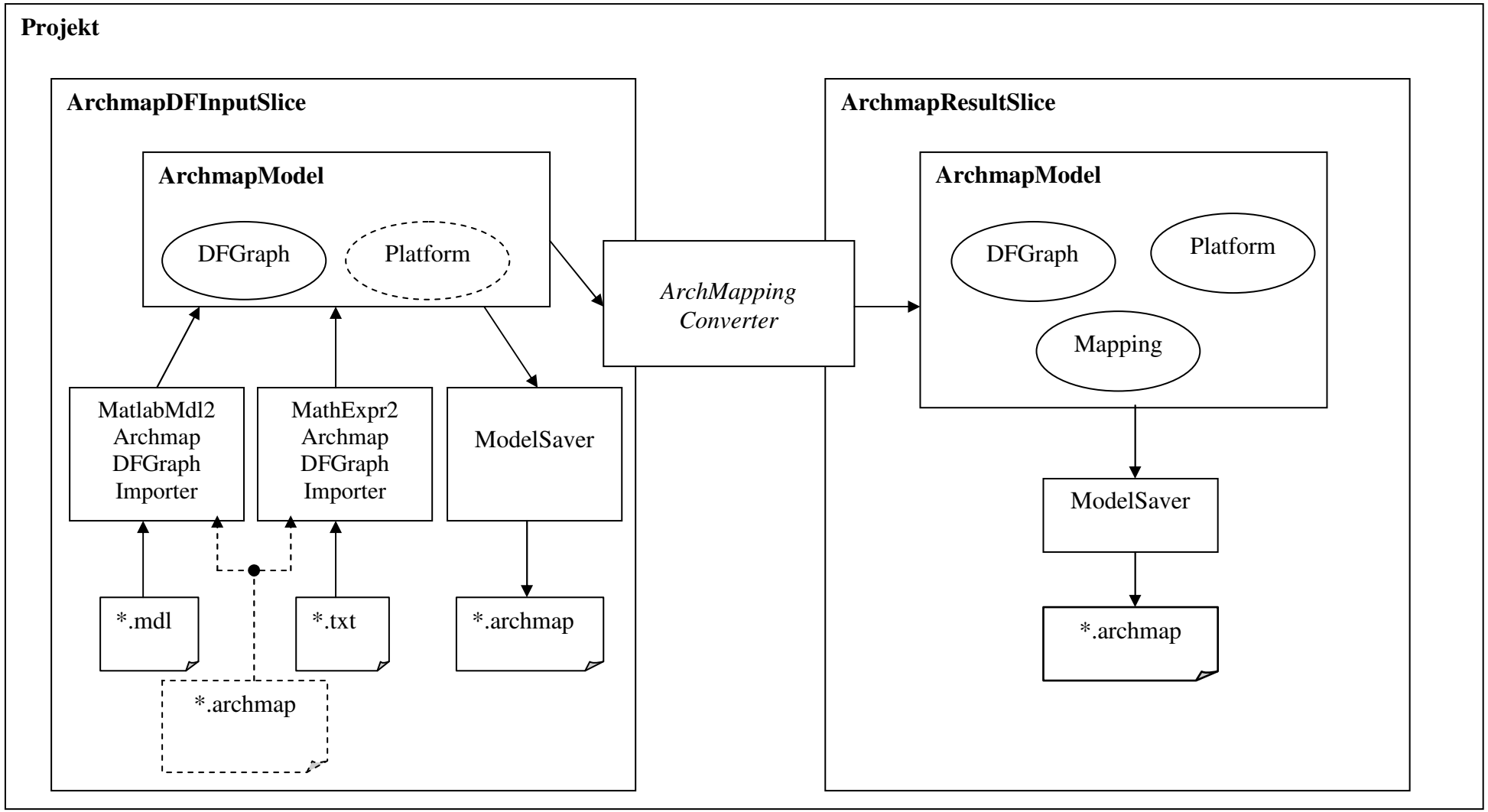


Abb. 3.3. Applikationsstruktur

Aufbaus von diesen und die Terminologie davon werden im Kapitel 4 erläutert. Es besteht auch die Möglichkeit die komplexen Datenstrukturen, die für die weitere Modellkonvertierung nötig sind, zu laden, wie z.B. die Plattformarchitektur des im Kapitel 2 erwähnten MDSP- oder des anderen Systems.

Es wird für die beiden Slices der gleiche Modellsaver benutzt, welcher die von der Applikation erzeugten Modelldaten im XML-ähnlichen Format (*.archmap) speichert. Der Benutzer kann dieses Format mit Hilfe von dem im Framework eingebauten *Modelleditor* als eine Baumstruktur ansehen und damit die Modelle analysieren.

3.2.2. Zielmodell

In diesem Punkt wird das in dieser Masterarbeit als Zielformat (*.archmap) für die Transformierung der Eingangsdaten (*.mdl, *.txt) verwendete Modell beschrieben. Dieses Modell wurde von Herrn M. Müllers Arbeitsgruppe (TU Ilmenau) entwickelt und besteht aus 3 Submodellen, die weiter betrachtet werden.

3.2.2.1. Graphenmetamodell *DFGraph*

Das Graphenmodell *DFGraph* stellt dem Benutzer der Applikation den Mechanismus für die Vorstellung der Eingangsdaten in Form eines Graphen zur Verfügung. Solcher Graph enthält 2 Arten von Knoten, das sind Operationsknoten und Datenknoten. Jeder Knoten hat eine Menge von spezifischen Eigenschaften, die im folgenden UML-Klassendiagramm (Abb. 3.4) dargestellt sind.

Am obersten Niveau dieses Modells ist die Klasse *DFGraph* und das ist das Modell, d.h. der ganze aus den Eingansdaten erzeugte Graph, selbst. Dieser Graph besteht aus einer beliebigen Anzahl (0..*) von Operationsknoten (*Node*) und Datenknoten (*Datum*), zwischen denen 4 Beziehungsarten definiert werden, die im Detail in der Tabelle 3.1 betrachtet sind.

Jeder Operations- und Datenknoten des Graphen hat seine eigene ID vom Typ *String*. Die Bedeutungen und die Werte dieser IDs sind von der Art der Eingansdaten abhängig und sind in den jeweiligen Punkten dieser Arbeit (4.3, 4.5) erklärt.

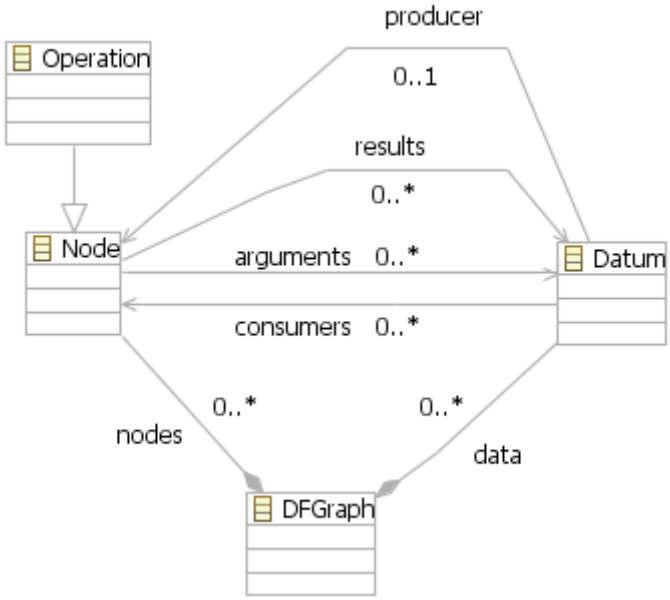


Abb. 3.4. UML-Klassendiagramm des Graphenmodells

Tabelle 3.1

Beziehungen zwischen den Klassen des Graphenmodells

Beziehung, Beziehungsrichtung, Vielfachheit	**Kommentar**
Producer, Datum → Node, 0..1	Jeder Datenknoten wird von maximal einem Operationsknoten erzeugt.
Results, Node → Datum, 0..*	Umgekehrte Beziehung. Jeder Operationsknoten kann einen oder mehrere Datenknoten erzeugen.
Arguments, Node → Datum, 0..*	Ein Operationsknoten nimmt als Argumente einen oder mehrere Datenknoten. Es gibt auch die Operationsknoten, die keine Datenknoten als Argumente nehmen.
Consumers, Datum → Node, 0..*	Umgekehrte Beziehung. Die Datenknoten werden von Operationsknoten als Argumente konsumiert.

3.2.2.2. Plattformmodell *Platform*

Das Plattformmodell *Platform* stellt das Verfahren für die vereinfachte Darstellung jedes Rechnersystems mit CPU-Bus-Architektur als ein Programmmodell vor. Das UML-Klassendiagramm des Plattformmodells ist in Abb. 3.5. dargestellt.

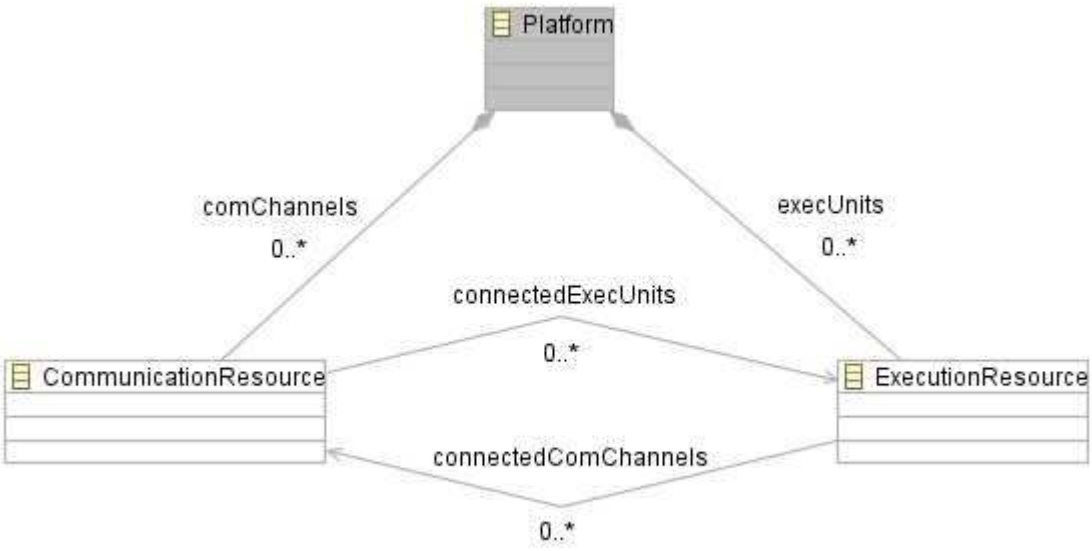


Abb. 3.5. UML-Klassendiagramm des Plattformmodells

Die Plattform des parallelen Rechnersystems enthält 2 Ressourcentypen, das sind die Rechnerressourcen oder Prozessoren (*ExecutionResource*) und die Kommunikationsressourcen (*CommunicationResource*). Jeder Prozessor ist mit genau der Anzahl von Bussen verbunden, die bei den Entwicklern des Rechnersystems vorgesehen wird.

Die Hierarchie und die Eigenschaften des Speichers des Rechnersystems bleiben leider in diesem Modell außer Betracht. Das kann man als die Schwachstelle dieses Modells interpretieren und diese kann durch die unterschiedliche Berücksichtigung der Übertragungszeiten zwischen den verschiedenen Speichertypen (z.B. Cache Memory, Shared Memory, Distributed Memory usw.) bei der Durchführung der Partitionierung behoben werden.

3.2.2.3. Partitionierungsmodell *Mapping*

Das Partitionierungsmodell *Mapping* realisiert die Abbildung von den Knoten des Graphenmodells *DFGraph* auf die Ressourcen des Zielrechnersystems, das in Form von dem Plattformmodell *Platform* dargestellt ist. In Abb. 3.6 ist die Struktur dieses Modells vorgestellt.

Die gesamte Abbildung besteht aus 2 Teilen, diese sind 2 Partitionierungsmodelle *PlatformElementMapping* und *DataTransferMapping*. Das erste Modell definiert, welche Operationsknoten auf welchen Prozessoren ausgeführt werden. Das Zweite bildet seinerseits die Verhältnisse zwischen den Datenknoten des Graphen und den Bussen des Rechnersystems, durch die die Daten der jeweiligen Datenknoten übertragen werden müssen. Mit den 1 zu 1 Beziehungen (1..1) im UML-Diagramm (Abb. 3.6) ist gemeint, dass jeder Knoten des Graphenmodells genau eine Abbildung haben muss.

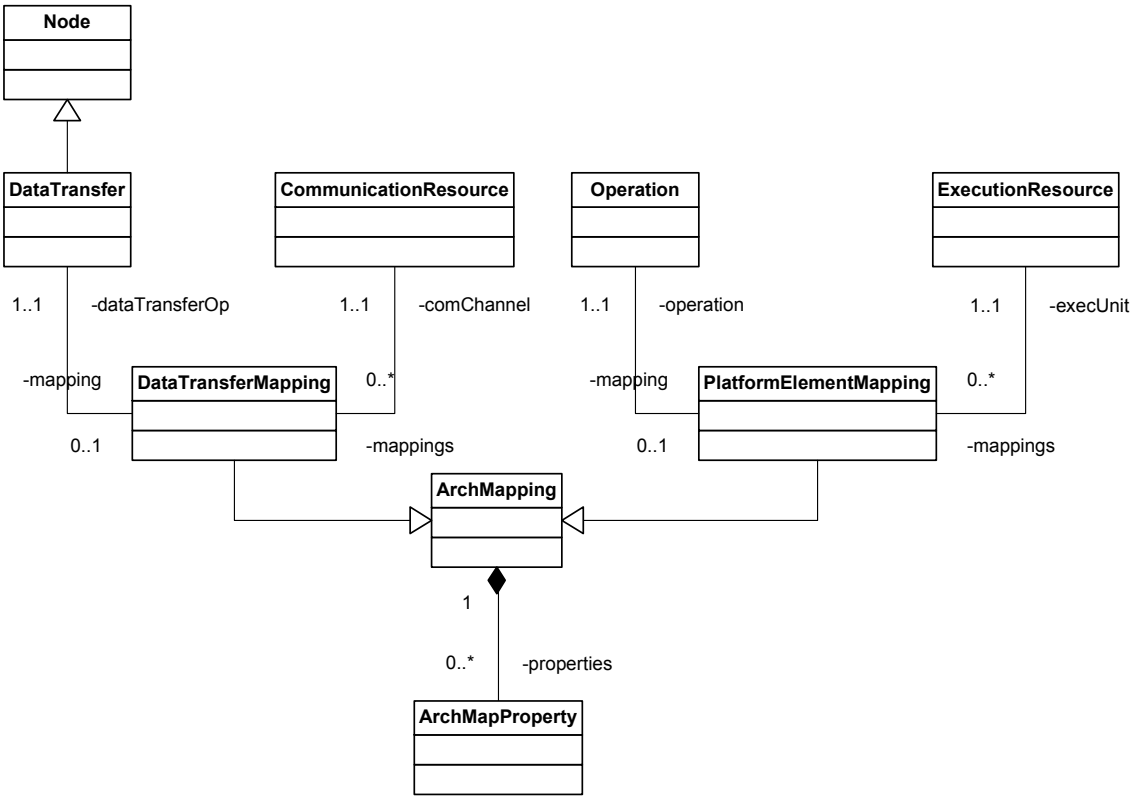


Abb. 3.6. UML-Klassendiagramm des Partitionierungsmodells

Im Partitionierungsmodell besteht auch die Möglichkeit, die Eigenschaften (Properties), d.h. in diesem Fall die Daten, die für die Partitionierung nötig sind, durch die abstrakte Klasse *ArchMapProperty* zu bestimmen. Im Rahmen der in dieser Arbeit entwickelten Applikation wird nur ein *Property* definiert, das ist die Berechnungszeit (in Takten oder ms) für die Operationsknoten des Graphen *(ExecDurationProperty).* Die Properties sind von dem Graphenmodell getrennt, weil sie für jedes für die Partitionierung verwendete Plattformmodell unterschiedlich sind, z.B. hängt die Berechnungszeit davon

ab, welche Frequenzen die Prozessoren des Zielrechnersystems haben. Die Datenübertragungszeiten zwischen den Prozessoren werden aus den Datengrößen berechnet, die von der Plattform unabhändig sind und deshalb direkt in den Attributen der Datenknoten festgelegt.

Die Erzeugung solcher Partitionierungsmodelle ist das Endziel dieser Masterarbeit und der Algorithmus dafür soll im Modellkonverter (Abb. 3.3) realisiert werden. Im Folgenden werden die Begriffe *Partitionierung* und *Mapping* als Synonyme verwendet.

3.3. Kozept des Modellkonverters

Der Modellkonverter *ArchMappingConverter* (MK) soll im Rahmen dieser Arbeit im Programmcode als ein Teil der enwickelten Applikation realisiert werden und die folgenden Anforderungen entsprechen:

1) Der MK muss das Eingangsmodell *des ArchmapDFInputSlices*, das die Beschreibung des Graphen und der Zielplattform enthält, ins Zielmodell *des ArchmapResultSlices,* das aus dem Eingangsmodell besteht, welches mit dem Mapping erweitert ist, transformieren.

2) Der MK muss den im Rahmen dieser Masterarbeit entwickelten und im Kapitel 4 beschriebenen Partitionierungsalgorithmus ausführen.

3) Der MK muss die qualitative und quantitative Analyse des Graphenmodells und die Berechnung der Modellparameter durchführen.

Die Beschreibungen und die Definitionen der für die Modellanalyse wichtigen Parameter werden weiter in diesem Kapitel präsentiert. Der gesamte Prozess der Implementierung des Modellkonverters ist seinerseits im Kapitel 4 betrachtet.

3.3.1. Qualitative und quantitative Analyse der Graphenmodelle

In diesem Punkt wird für die Bestimmung der Definitionen der für die Analyse wichtigen Parameter das folgende vereinfachte Graphenmodell verwendet. Die Knoten des Graphen stellen die Operationen (die Teile des gesamten Programmablaufs) vor, die ohne Unterbrechung auf den Prozessoren des Zielrechnersystems mit den Eingangsdaten ausgeführt werden müssen. Die Kanten sind ihrerseits die Datenabhängigkeiten zwischen

den Operationen. Jeder Knoten und jede Kante des Graphen hat seine eigene Ausführungszeit (in Takten des Prozessors), die man als die Operations- bzw. Übertragungsdauer interpretiert und im Graphenmodell als ganze positive Zahl darstellt. Während der Ausführung einer Operation oder der Datenübertragung sind die jeweiligen Prozessoren bzw. Busse des Rechnersystems besetzt. Wenn es in einem solchen Graphenmodell keine Zyklen gibt, dann wird dieser Graph *die parallele Schichtform* (Abb. 3.7) genannt [5].

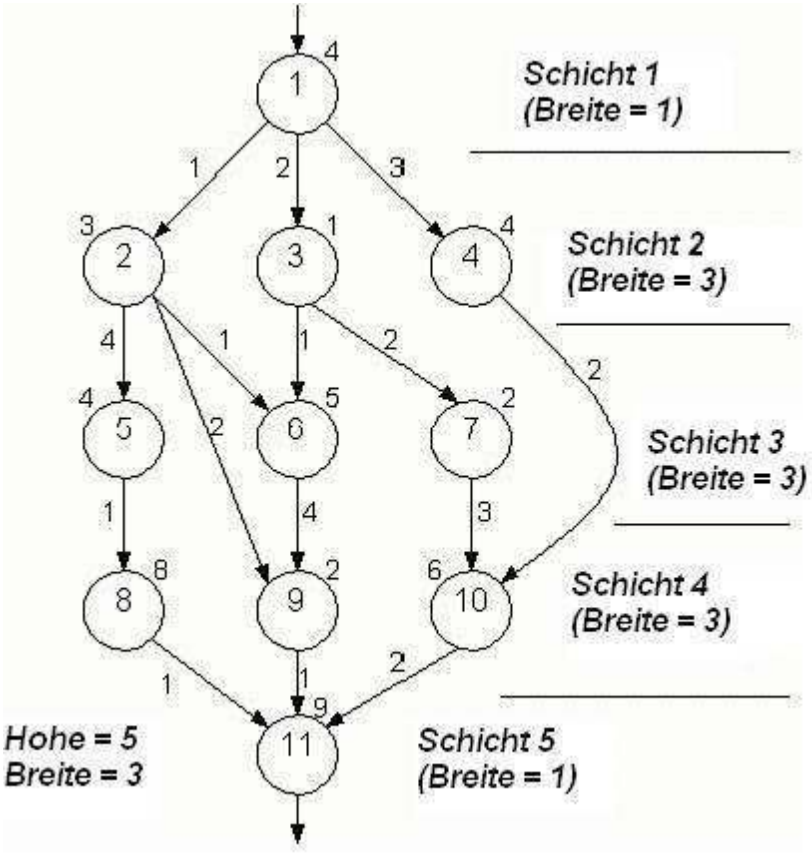


Abb. 3.7. Beispiel der parallelen Schichtform

Die parallele Schichtform kann man auch als ein mathematisches Objekt definieren, das unabhängig von dem Zielrechnersystem ist. Alle Knoten des azyklischen Graphen werden in die durchnumerierten Teilmengen V_i (i=1 .. M, M$\in N^+$) vereinigt, so dass, wenn die Kante *e* von dem Knoten $v_1 \in V_j$ zum Knoten $v_2 \in V_k$ führt, dann ist *j*<*k*. Jede Teilmenge V_i nennt man *eine Schicht* und *i* ist die Nummer von dieser. Die Anzahl der Knoten in der Schicht definiert man als die *Schichtbreite*. Die maximale

Schichtnummer der parallelen Schichtform bestimmt ihre *Höhe* und die maximale Breite der Schichten – ihre *Breite*.

Die Operationen der gleichen Schichten sind von einander unabhängig und können parallel ausgeführt werden. Deshalb stellt *die Breite* der parallelen Schichtform die notwendige Anzahl der Prozessoren für die schnellste Ausführung dieses Graphen dar und *die Höhe* repräsentiert die minimale gesamte Ausführungzeit (in Anzahl der längsten Operationen).

Der kritische Pfad des Graphen ist die Reihenfolge der Knoten, die mit dem Eingangsknoten des Graphen (mit der Wurzelknoten des Graphen) anfängt, sowie sich mit dem Ausgangsknoten (mit dem Blatt des Graphen) endet und genau einen Knoten aus jeder Schicht enthält, für den die minimal mögliche Startzeit der Ausführung $T_{min\ i}$ (3.1) und die maximal mögliche Startzeit der Ausführung $T_{max\ i}$ (3.2) gleich sind [5].

$$T_{min\ i} = \max_{j=1,\dots s} (T_{min\ j} + t_j + T_{ji}) \qquad (3.1);$$

$$T_{max\ i} = \min_{r=1,\dots v} (T_{max\ r} - t_i - T_{ir}) \qquad (3.2),$$

dabei entspricht *s* der Anzahl des Vorgängers des Knotens *i*; *v* der Anzahl des Nachfolgers des Knotens *i*; t_j *(*t_i*)* der Ausführungszeit des Knotens *j (i)* und T_{ji} (T_{ir}) der Übertragungszeit zwischen den Knoten *j* und *i* (*i* und *r*).

Der Graph kann mehr als nur einen kritischen Pfad enthalten.

Die Suche nach dem kritischen Pfad ist für die Berechnung des wichtigen Parameters des Graphenmodells T_{min} nötig, der die minimal mögliche Ausführungszeit des Graphen auf dem Rechnersystem mit unbegrenzten Ressourcen vorstellt. T_{min} ist die Zeitlänge des kritischen Pfades. Der maximale Speedup $Speedup_{max}$, den man für den gegebenen Graphen mit jedem möglichen Rechnersystem erreichen kann, ist der Formel (3.3) gemäß zu berechnen:

$$Speedup_{max} = T_{max} / T_{min} \qquad (3.3),$$

dabei entspricht T_{max} der Ausführungszeit des Graphen auf dem Rechnersystem, der nur aus einem Prozessor besteht (sequentielles Rechnersystem).

Den maximalen Speedup kann man auch mit Hilfe von den *amdahlschen Gesetzen* bewerten (benannt 1967 nach Gene Amdahl). Nach Amdahl wird der Speedupzuwachs vor

allem durch den sequentiellen Anteil des Problems, in diesem Fall des Graphen, beschränkt, da sich dessen Ausführungszeit durch Parallelisierung nicht verringern lässt. Im Folgenden sind 3 *amdahlschen Gesetze* dem [6] entsprechend angegeben.

Erstes Amdahlsches Gesetz. Die Produktivität des Rechnersystems, das aus den miteinander verbundenen Komponenten (Prozessoren) besteht, ist im allgemeinen Fall von der langsamsten Komponente (Prozessor) abhängig.

Zweites Amdahlsches Gesetz. Möge das Rechnersystem aus *s* gleichen universellen einfachen Komponenten (Prozessoren) bestehen, die alle bei der Ausführung des parallelen Anteils des Problems belastet sind. Dann ist der maximale Speedup *R* der Formel (3.4) entsprechend zu berechnen:

$$R = s / (\beta s + (1-\beta)) \qquad (3.4),$$

dabei entspricht $\beta=k/N$ dem sequentiellen Anteil des Problems, *N* der gesamten Anzahl der Operationen und *k* der Anzahl der sequentiellen Operationen.

Drittes Amdahlsches Gesetz. Möge das Rechnersystem aus den gleichen universellen einfachen Komponenten (Prozessoren) bestehen. Für jeden beliebigen Betriebsmodus des Rechnersystems ist sein Speedup mit dem Wert $1/\beta$ begrenzt.

Die Graphen der Probleme, die mit den parallelen Schichtformen dargestellt sind, können die verschiedenen Verhältnisse zwischen der gesamten Ausführungszeit (t_a) und Datenübertragungszeit ($t_ü$) haben. Dieser Verhältnisse gemäß unterscheidet man die *schwach* ($t_a >> t_ü$), *mittel* ($t_a \approx t_ü$) und *stark gebundenen* ($t_a << t_ü$) Graphen.

Für jeden Graphentyp spielt die richtige Auswahl der Verhältnisse zwischen der Anzahl der Prozessoren und Busse des Zielrechnersystems die Hauptrolle für die Ausführung des Problems in kürzester Zeit. Für die *schwach gebundenen* Graphen muss man mit Rücksicht auf alle oben erwähnten theoretischen Grundlagen für qualitative und quantitative Analyse der Graphenmodelle die passende Anzahl der Prozessoren auswählen. Für die *stark gebundenen* Graphen ist ihrerseits die Anzahl der Busse im Zielrechnersystem wichtig und für die *mittel gebundenen* das ausgewogene Verhältnis zwischen der Anzahl der Prozessoren und Busse.

Kapitel 4

Implementierung der Applikation

In diesem Kapitel wird die Implementierung der Applikationselemente, die im vorigen Kapitel nur als Konzepte vorgestellt wurden, beschrieben. Zunächst sollen jedoch einige Überlegungen vorangestellt werden, die sich der Betrachtung zur Verfügung stehender Software, Programmumgebung und Werkzeuge für Programmierung widmet, um zu analysieren, welche Möglichkeiten für die Realisierung der entstehenden Applikation bestehen, und inwiefern flexibel und veränderbar die künftige Applikationsstruktur sein wird. Daran anschliessend werden die Implementierungen von jedem Applikationselement, das sind die Parser und der Modellkonverter, detailliert erläutert.

4.1. Eclipse als Entwicklungsumgebung

Eine der Anforderungen an die Implementation der Applikation ist die Entwicklung von dieser in Java. Als Entwicklungsumgebung soll *Eclipse* benutzt werden. In folgenden 2 Punkten wird diese Entwicklungsumgebung und ein von ihrer Plug-ins, der für die Ausführung des Parsings nötig ist, kurz vorgestellt.

Eclipse ist ein quelloffenes (Open-Source) Programmierwerkzeug zur Entwicklung von Software. Ursprünglich wurde Eclipse als Entwicklungsumgebung für die Programmiersprache Java benutzt, aber mittlerweile wird es aufgrund seiner Erweiterbarkeit auch für viele andere Entwicklungsaufgaben und Programmiersprachen eingesetzt [7].

Bis einschließlich zur Version 2.1 war Eclipse als erweiterbare integrierte Entwicklungsumgebung (IDE) konzipiert. Seit Version 3.0 ist Eclipse selbst nur der Kern, der die einzelnen Plug-ins lädt, die dann die eigentliche Funktionalität zur Verfügung

stellen. Die Abhängigkeiten zu anderen Plugins, die z.B. für die Funktionalität der im Punkt 3.2 beschriebenen Modelle nötig sind, werden in einer speziellen Datei *plugin.xml* im Eclipse-Projekt für die Applikation beschrieben. Die verfügbaren Plugins werden beim Start des Eclipses zusammengetragen und zusammengefügt, der eigentliche Code wird aber erst geladen, wenn er wirklich benötigt wird.

Alle detaillierten Informationen zu Eclipse und seinen Fähigkeiten sowie Eclipse-eigene Dokumentation und die einschlägige Fachliteratur sind auf der Web-Seite *www.eclipse.org* erläutert.

Eclipse ist plattformunabhängig (betriebsystemunabhängig). In dieser Arbeit wird die zur Zeit letzte aktuelle Version von Eclipse, das ist Eclipse 3.5 Galileo, für die Applikationsentwicklung unter MS Windows benutzt.

4.2. Eingangsdatenparsing mit JavaCC

Ein *Parser* ist ein Computerprogramm, das in der Computertechnik für die Zerlegung und Umwandlung einer beliebigen Eingabe in ein für die Weiterverarbeitung brauchbares Format zuständig ist. Der Parser gibt auch die Analyse einer Eingabe in einer gewünschten Form aus und erzeugt zusätzlich Strukturbeschreibungen [8].

Im Rahmen dieser Arbeit werden 2 Arten von Eingabedaten, das sind die mathematischen Formeln und die Modelldateien von Simulink, und ein Zielformat, das ist das in Punkt 3.2.2.1 erwähnte Graphenmodell, für die Syntaxanalyse (*Parsing*) verwendet.

Der für die Analyse des Texts verwendete Parser ist in der Regel ein lexikalischer Scanner. Dieser zerlegt die Eingabedaten in Tokens (Eingabesymbole bzw. „Wörter“, die der Parser versteht). Die Zerlegung in Tokens folgt einer regulären Grammatik (grammatischen Regeln), deshalb ist der Scanner meist ein endlicher Automat und die Tokens dienen als atomare Eingabezeichen von diesem.

Es gibt verschiedene Parsertypen. Man kann die nach der Reihenfolge, in der die Knoten des Ableitungsbaums (*parse tree*) erstellt werden (top-down, bottom-up oder left corner), spezifischer Vorgehensweise (LL, LR, SLR, LALR, LC u.a.), Implementierungstechnik (rekursiv absteigend, rekursiv aufsteigend oder tabellengesteuert) oder Grammatikart (Parser für kontextfreie Grammatiken, Parser für kontextsensitive Grammatiken) unterscheiden.

Die algemeine Struktur des Parsers ist in Abb. 4.1 dargestellt.

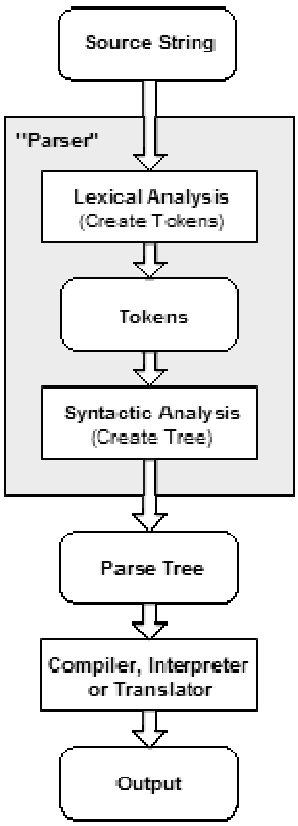


Abb. 4.1. Allgemeine Struktur des Parsers

In dieser Arbeit wird JavaCC, ein Plug-in für Eclipse, als Generator für die Parser verwendet (https://javacc.dev.java.net/). Dieser *Parsergenerator* erzeugt im Java-Code einen top-down *LL(k)-Parser* für kontextfreie Grammatiken und das bedeutet, dass er die Eingabe von *l*inks nach rechts bearbeitet, um eine *L*inksableitung der Eingabe zu berechnen, und während des Parsings *k* Tokens vorausschauen kann. JavaCC ist Open Source und unter den Bedingungen der Berkeley Software Distribution (BSD) Lizenz herausgegeben. Die Tokens und die Grammatikregeln des Parsers müssen in *Backus–Naur Form (BNF)* dargestellt werden. Die Prioritäten von Tokens und Grammatikregeln sind von der Reihenfolge, in der man die im Parsertext schreibt, abhängig, d.h. je früher das Token bzw. die Grammatikregel im Parser definiert ist, desto höher ist die Priorität von diesen. Der Parser vergleicht die Eingabedaten mit den Grammatikregeln nach Äquivalenz ihren Prioritäten entsprechend. JavaCC bietet auch eine Möglichkeit die Tokens den *lexikalen Zuständen* (lexical states) gemäß zuzuordnen. Solche Zustände nennt man auch *Tokenmodi* oder Tokenfarben. Jeder Tokenmodus hat seinen eigenen Identifikationsname.

Wenn keine lexikalen Zustände definiert wurden, läuft der Parser automatisch im von JavaCC-Tokenmanager generierten *DEFAULT* Modus. Für jeden Tokenmodus sind 4 Arten von Grammatikregeln festgelegt, die in der Tabelle 4.1 dargestellt sind. Der Tokenmanager kann sich gleichzeitig nur in einem lexikalen Zustand befinden [9].

Tabelle 4.1

Arten von JavaCC Grammatikregeln

Artname im JavaCC-Programm	**Erläuterung**	**Beispiel**

TOKEN	Das Token wird dem gegebenen String entsprechend erzeugt und zum Parser übertragen	<HTML_STATE> TOKEN :
		{
		<PHP_BEGIN: "<?" ("php")?>|
		<PHP_EXPR: "<?=" > |
		<HTML_OTHER: "<" ~[] >
		}

SPECIAL_TOKEN	Das Specialtoken wird erzeugt, der nicht an dem Prozess des Parsings teilnimmt	SPECIAL_TOKEN :
		{
		<SINGLE_LINE_COMMENT:
		"//" (~["\n","\r"])*
		("\n"|"\r"|"\r\n")>
		}

SKIP	Der gegebene String wird beim Parser ignoriert	SKIP :
		{
		<COMMENT_START: "/*"> :
		WithinComment
		}
		
		<WithinComment> SKIP :
		{
		<COMMENT_END: "*/"> :
		DEFAULT
		}

MORE	Der Prozess des Parsings wird fortgesetzt.
	Der gegebene String wird als ein Präfix zum nächsten vom Parser verglichenen String hinzugefügt
		<WithinComment> MORE :
		{
		<~[]>
		}

4.3. Parser der mathematischen Formeln

Wie bereits zuvor erwähnt, muss man, um einen Parser zu bestimmen, die Tokens und die Grammatik definieren. Die Tokens für die mathematischen Formeln sind in Form von mathematischen Operationen, in diesem Fall Addition (A), Subtraktion (S), Multiplikation (M), Division (D), Potenz (P), Sinus (si) und Kosinus (co), und Operanden von diesen, d.h. Konstanten und Variablen, dargestellt. Die Formeln dürfen auch eine beliebige Anzahl von Klammern enthalten. Alle Operationen, außer Sinus und Kosinus, werden im Folgenden nur als binäre Operationen für das Parsing betrachtet. Die grammatischen Regeln des Parsers werden den Prioritäten der Operationen gemäß definiert.

Der Nutzer gibt dem Parser die mathematische Formel wie einen String ein und danach wird daraus ein Graph erzeugt, der dem im Punkt 3.2.2.1 erwähnten Graphenmodell entspricht. Jede Operation wird in einen Operationsknoten transformiert sowie jeder Operand in einen Datenknoten. Die Beziehungen zwischen den Knoten werden der mathematischen Syntax (Argumente – Operation – Ergebnis) entsprechend bestimmt. Unten sind die im Parser verwendeten Tokens (Tabelle 4.2) sowie das graphische Ergebnis der Transformierung der Formel $F=\sin\left((8+a)^{3^{b-4}}\right)*6.55$ (Abb. 4.2) angegeben.

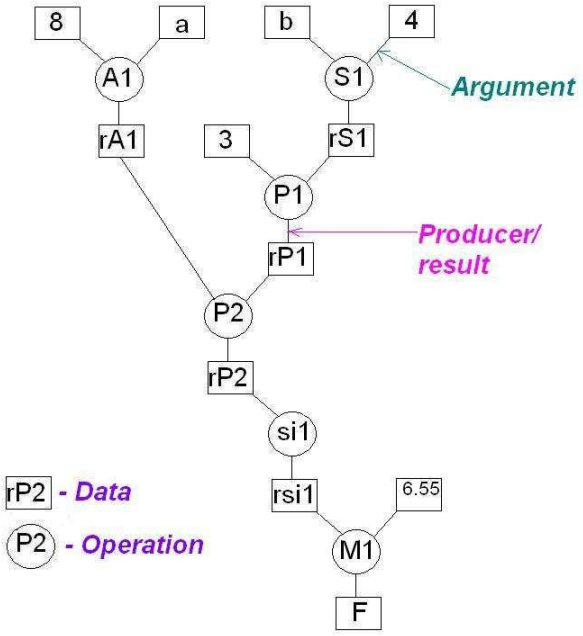


Abb. 4.2. Graphisches Ergebnis der Transformierung der Formel $F=\sin\left((8+a)^{3^{b-4}}\right)*6.55$

Tabelle 4.2

Die Tokens des Parsers der mathematischen Formeln

Token	Kommentar
Mathematische Operationen	
PLUS: "+"	Addition
MINUS: "-"	Subtraktion
MULTIPLY: "*"	Multiplikation
DIVIDE: "/"	Division
POTENZ: "^"	Potenz
Operanden	
NUMBER: (<DIGIT>)+ ("." (<DIGIT>)+)? DIGIT: ["0"-"9"]	Konstanten
VARIABLE: (<LETTER>)+ ((<DIGIT>)+)? LETTER: ["a"-"z"] | ["A"-"Z"] DIGIT: ["0"-"9"]	Variablen

Die grammatischen Regeln des Parsers sind dem folgenden Muster gemäß aufgebaut (4.1):

$$FUNKTION_i() :: FUNKTION_{i+1}() (<OP\_TOKEN_i> FUNKTION_i())^* \qquad (4.1),$$

dabei entspricht $FUNKTION_i()$ der Funktion des Parsers für Operation von Priorität i und $<OP\_TOKEN_i>$ dem Token dieser Operation (siehe Tabelle 4.2). Die Prioritäten der Operationen sind in der folgenden Liste dargestellt. Je kleiner die Operationsnummer ist, desto höher ist die Priorität der jeweiligen Operation für den Parser:

1. Addition (+)
2. Subtraktion (-)
3. Multiplikation (*)
4. Division (/)
5. Potenz (^)
6. sin( ) / cos( ) / ( ) / Operanden

Der Parser liest die vom Nutzer angegebene mathematische Formel von links nach rechts aus, vergleicht die Teile von dieser mit den grammatischen Regeln von 1. bis 6. und bildet den Ableitungsbaum sowie den Graphen. Für jeden Ausdruck in Klammern macht er alle oben erwähnten Schritte wieder und verbindet den neu erzeugten Baum bzw. Graphen mit bereits Gebildetem. Dieser Prozess wird solange durchgeführt bis der ganze String von dem Parser bearbeitet wurde. Als Ergebnis wird am Ende die graphische Darstellung dieser

mathematischen Formel erzeugt, die dem im Punkt 3.2.2.1 beschriebenen Graphenmodell entspricht.

4.4. Simulink Modellbeschreibungsformat (MDL)

MoDeL (MDL) Format stellt das Simulink Modellbeschreibungsformat vor. Die MDL-Datei ist die strukturierte ASCII Datei, die die Paare der Schlüsselwörter und der jeweiligen Werte der Parameter für die Beschreibung des Modells enthält. Die Modellkomponenten werden in der hierarchischen Reihenfolge beschrieben. Die logischen Blöcke der Komponenten in der MDL-Datei werden mit Hilfe von geschweiften Klammern voneinander getrennt.

In Abbildung 4.3 ist die allgemeine Struktur der Datei dargestellt. Die Sektion *Model* befindet sich auf dem obersten hierarchischen Niveau der MDL-Datei und enthält alle anderen Sektionen sowie die Werte der Parameter der Modellebene, unter diesen die Name des Modells, die letzte für die Öffnung der Datei benutzte Version der Simulinksoftware und die Parameter für die Modellkonfiguration. Die *Simulink.ConfigSet* Sektion definiert die aktive Modellkonfiguration. Die *BlockDefaults* und *BlockParameterDefaults* bestimmen die für das ganze Modell gemeinsamen Eigenschaften der Blöcke, die aber von den Parametern der einzelnen Blöcke in der Sektion *System* ungültig gemacht werden können. Jede Subsektion *Block* definiert die gemeinsamen Eigenschaften von jedem einzelnen im Modell verwendeten Blocktyp. Die *AnnotationDefaults* Sektion enthält die Standardwerte der allen Annotationen (die von dem Nutzer bestimmten Kommentare zu den Komponenten) [10].

Das System des obersten Niveaus und jedes Subsystem des Modells wird in den einzelnen Sektionen *System* beschrieben. Jede solche Sektion bestimmt die Parameter des Systemniveaus und enthält *Block, Line* und *Annotation* Sektionen für jeden Block, Line und Annotation des Systems. Jede Line, die eine Verzweigungsstelle hat, enthält auch eine Sektion *Branch*, eine pro Zweig.

Die Sektionen *Block* und *Line* sind die wichtigsten Sektionen der MDL-Datei, weil mit Hilfe von diesen die Verhältnisse zwischen allen Komponenten des Modells dargestellt werden. Deshalb sollen diese Sektionen für die Parseranalyse, d.h. für den Aufbau des im Punkt 3.2.2.1 erwähnten Graphenmodells, benutzt werden.

```
    Model {
      <Model Parameter Name> <Model Parameter Value>
      ...
      Array {
        Simulink.ConfigSet {
          $ObjectID <Object ID>
          <ConfigSet Parameter Name> <ConfigSet Parameter Value>
          ...
        }
      }
      Simulink.ConfigSet {
        $PropName "ActiveConfigurationSet"
        $ObjectID <Object ID>
      }
      BlockDefaults {
        <Block Parameter Name> <Block Parameter Value>
        ...
      }
      BlockParameterDefaults {
        Block {
          <Block Parameter Name> <Block Parameter Value>
          ...
        }
      }
      AnnotationDefaults {
        <Annotation Parameter Name> <Annotation Parameter Value>
        ...
      }
      LineDefaults {
        <Line Parameter Name> <Line Parameter Value>
        ...
      }
      System {
        <System Parameter Name> <System Parameter Value>
        ...
        Block {
          <Block Parameter Name> <Block Parameter Value>
          ...
        }
        Line {
          <Line Parameter Name> <Line Parameter Value>
          ...
          Branch {
            <Branch Parameter Name> <Branch Parameter Value>
            ...
          }
        }
        Annotation {
          <Annotation Parameter Name> <Annotation Parameter Value>
          ...
        }
  }
    }
```

Abb. 4.3. Allgemeine Struktur der MDL-Datei

4.5. Parser der MDL-Dateien

Wie es früher erzählt wird, werden für die Parseranalyse nur die Sektionen *Block* und *Line* betrachtet. Die MDL-Datei hat eine große Menge von Blocktypen, die man in 14 Gruppen einteilen kann. Diese sind kontinuierliche Blöcke (Integrator, PID Controller u.a.), diskontinuierliche Blöcke (Relay, Saturation u.a.), diskret Blöcke (Memory, Unit Delay u.a.), Logik- und Bitoperationen (Bit Set, Logical Operator, Shift Arithmetic u.a.), LookUp Tabellen, mathematische Operationen, Modellverifikationen (Assertion, Check Static Range u.ä.), Model-Wide Utilities (Block Support Table u.ä.), Ports & Subsysteme (Subsystem, Outport, Inport u.a.), Signalattribute (Bus to Vector, IC, Probe u.a.), Signalrouting (Bus Selector, Bus Creator u.ä.), Sinks (Display, Terminator u.a.), Sourcen (Clock, Constant) und von dem Nutzer definierte Funktionen (z.B. S-Funktion) [11].

Um das Parsing der MDL-Datei zu vereinfachen, werden weiter nur die Blöcke von Typ *Ports & Subsystems* sowie *Lines* betrachtet. Die Subsysteme werden im Graphenmodell die Operationsknoten vorstellen und die Ausgangsports von diesen werden in Datenknoten transformiert, die Lines werden ihrerseits die Verbindungen zwischen den Daten- und Operationsknoten definieren.

Vor dem Parsing ist es auch notwendig ein Subprogramm zu entwickeln, das die MDL- in eine Textdatei umwandelt und dabei die Sektionen der MDL-Datei löscht, die man für die Analyse nicht braucht, d.h. alle Sektionen außer der *System* Sektion und ihrem Inhalt. Dieses Subprogramm sucht nach den 2 Schlüsselwörtern in der MDL-Datei, das sind das erste gefundene Schlüsselwort *System {* und Schlüsselwort *# Finite State Machines*, das eventuell nach der *System* Sektion in der MDL-Datei anwesend sein kann und das die Anfangszustände der Finitautomaten des Simulinkmodells definiert, und danach bildet die Textdatei mit dem von dem Nutzer definierten Namen, die nur den Inhalt der obersten *System* Sektion enthält. Diese Textdatei wird als die Eingangsdatei (Eingangsdatenfluss) für den Parser benutzt.

Im JavaCC Parsergenerator besteht eine Möglichkeit die von dem Nutzer ausgewählten Tokenmodi zu bestimmen. Im entwickelten Parser sind 4 Tokenmodi definiert, das sind *NORMAL*, *IGNORE*, *LINES* und *KLAMMER* Modi. Für jeden Modus gibt es ein bestimmtes Set von Tokens, nach denen der Parser sucht. Alle Tokensets sind den Prioritäten entsprechend in der Tabelle 4.3 dargestellt.

Der Parser startet sich im *NORMAL* Modus, sucht nach dem Anfang der obersten *System* Sektion, der mit dem Token *System {* in der Datei definiert wird, und schaltet sich in den *IGNORE* Modus um. Die Aufgabe von dem *IGNORE* Modus ist die Suche nach den Blöcken und Lines und primäre Analyse von diesen. In diesem Modus muss der Parser entscheiden, ob er den Block bzw. Line für den Aufbau des Graphenmodells braucht, d.h. er prüft ob der Block von Typ Subsystem ist oder die Line tatsächlich eine Line ist. Falls ja, geht das Programm in *NORMAL* bzw. *LINES* Modus, sonst – in *KLAMMER* Modus.

Weiter wird im *NORMAL* Modus geprüft, ob das gegebene Subsystem atomar ist, d.h., dass vor der Benutzung dieses Parsers der Nutzer entscheiden muss, wie detailliert er sein Modell damit analysieren will. Um das zu bestimmen muss man die Subsysteme, die man analysieren möchte, als atomare Subsysteme im Similinkmodell definieren. Dafür gibt es das Schlüsselwort *TreatAsAtomicUnit*. Wenn es "on" ist, ist das Subsystem atomar und der Parser läuft im NORMAL Modus weiter. Falls es "off" ist oder das Schlüsselwort *TreatAsAtomicUnit* in der Beschreibung des Blocks fehlt, dann ignoriert der Parser diesen Block und den Inhalt von diesem mit Hilfe des *KLAMMER* Modus. Wenn das Subsystem atomarer Bestimmung entspricht, speichert der Parser seinen Namen in die „Datenbank“ (Array), erzeugt den Operationsknoten mit diesem Namen, sowie reserviert den Platz im Speicher für die Parameter des Outports dieses Blocks der Anzahl von diesen entsprechend. Die Suche nach den Outportblöcken wird im *IGNORE* Modus und die Analyse von diesen, d.h. die Speicherung der Namen in die „Datenbank“ (Matrix) dem Subsystem und der Outportnummer entsprechend, wieder im *NORMAL* Modus durchgeführt.

Im *LINES* Modus analysiert der Parser, wie die Subsysteme miteinander verbunden sind. Er bearbeitet die Linesektionen und sucht nach den Quellblöcken und Zielblöcken sowie nach den Ports von diesen. Wenn ein Quellblock bzw. Zielblock gefunden ist, wird überprüft ob er in der „Datenbank“ existiert. Falls nicht, wird der Inhalt dieser Linesektion mit Hilfe des *KLAMMER* Modus ignoriert, sonst werden die Verbindungen zwischen den Ports der jeweiligen Subsysteme hergestellt und die Datenknoten mit den Outportnamen des Quellblocks erzeugt, und es werden die *Argument-Producer* Beziehungen zwischen den Datenknoten und den Operationsknoten, die den Quellblöcken und Zielblöcken entsprechen, bestimmt.

Tabelle 4.3

Die Tokens für alle Parsermodi des MDL-Parsers

Token/Schlüsselwort*	Wert der Parameter	Typ Beispiel des Formats	BNF
NORMAL Modus			
System *{*	enthält alle anderen Sektionen	-	-
Name	Blockname	String "PID Regler"	NAME_ID:: ((<LETTER>)+ (<LETTER> | " ")* (<LETTER>)+) | (<LETTER>)+ LETTER:: ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_" | "-" | "\"" | "[" | "]"
Port	Nummer des Inports/Outports (fehlt beim Port #1)	string — {'1'} "2"	OUTPORT_NUM:: ("\"")(["0"-"9"])+("\"")
Ports	definiert, wie viele Ports von jedem Typ (Inport, Outport u.a.) der Block (Subsystem) hat	Vektor [2,2] *oder* [3] falls nur Inports	PORTS_ID1:: ("[") (["0"-"9"])+ (" ")* (",") (" ")* (["0"-"9"])+ ("]") *oder* PORTS_ID2:: ("[") (["0"-"9"])+ ("]")
Position	Position des Blocks auf dem Bildschirm	Koordinatenvektor (in Pixels) [1615, 596, 1655, 659]	POSITION_ID:: ("[") ((["0"-"9"])+ (" ")* (",") (" ")* (["0"-"9"])+)+ ("]")
Orientation	Orientierung des Blocks	{'right' | 'left' | 'up' | 'down'} "down"	NAME_ID:: ((<LETTER>)+ (<LETTER> | " ")* (<LETTER>)+) | (<LETTER>)+ LETTER:: ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_" | "-" | "\"" | "[" | "]"
TreatAsAtomicUnit	definiert, ob das Subsystem atomar ist	{'off' | 'on'} on	ON_OFF:: "on" | "off"
IGNORE Modus			
Block *{*	enthält Parameter der Blöcke (siehe NORMAL Modus)	-	-

* Die Schlüsselwörter enthalten keine geschweiften Klammern im Gegensatz zu den Tokens

Token/Schlüsselwort*	Wert der Parameter	Typ Beispiel des Formats	BNF
BlockType	Blocktyp	String SubSystem	BLOCK_TYPE:: "SubSystem" | "BusCreator" | "BusSelector" | "Constant" | "Outport" | "Inport" | "Terminator" | "Lookup" | "Product" | "Sum" | ...
Line *{*	enthält Parameter der Lines (siehe LINES Modus)	-	-
LINES Modus			
Name	Linename	String "Winkelfehler"	Port_ID:: ((<LETTERS>)+ (<LETTERS> | " ")* (<LETTERS>)+) | (<LETTERS>)+ LETTERS:: ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_" | "-" | "\"" | "[" | "]" | "<" | ">" | "."
SrcBlock	Quellblockname	String "PID Regler"	Port_ID:: ((<LETTERS>)+ (<LETTERS> | " ")* (<LETTERS>)+) | (<LETTERS>)+ LETTERS:: ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_" | "-" | "\"" | "[" | "]" | "<" | ">" | "."
SrcPort	Quellportnummer	Integer 1	PORT_NUM:: (["0"-"9"])+
Branch *{*	bestimmt die Verzweigungsstelle	-	-
DstBlock	Zielblockname	String "Anti Wind-Up"	Port_ID:: ((<LETTERS>)+ (<LETTERS> | " ")* (<LETTERS>)+) | (<LETTERS>)+ LETTERS:: ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_" | "-" | "\"" | "[" | "]" | "<" | ">" | "."
DstPort	Zielportnummer	Integer 2	PORT_NUM:: (["0"-"9"])+
KLAMMER Modus			
{	Klammer auf	-	KLAMMER_AUF:: "{"
}	Klammer zu	-	KLAMMER_ZU:: "}"

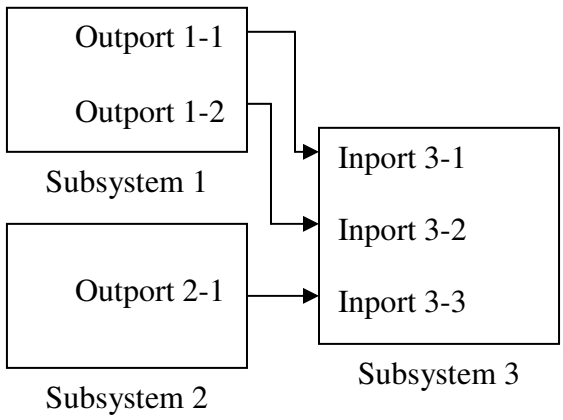
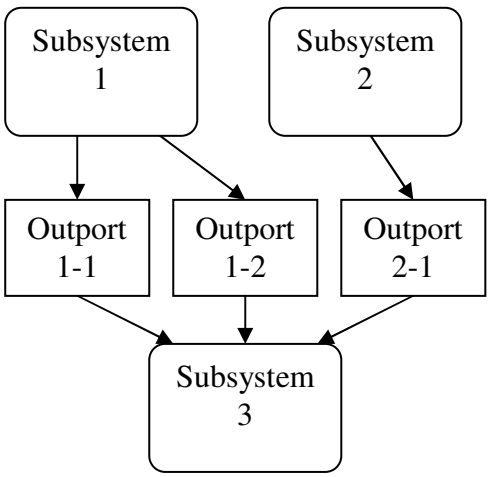
* Die Schlüsselwörter enthalten keine geschweiften Klammern im Gegensatz zu den Tokens

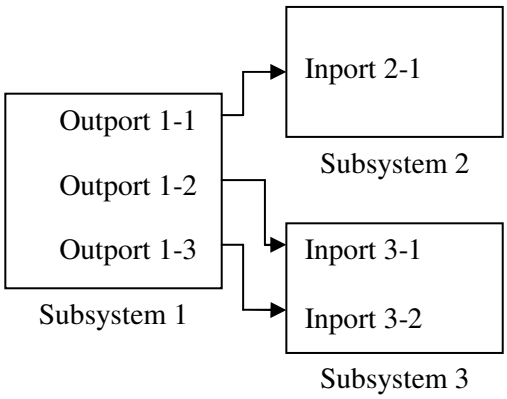
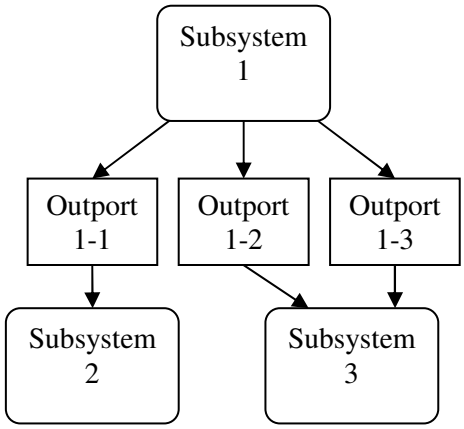
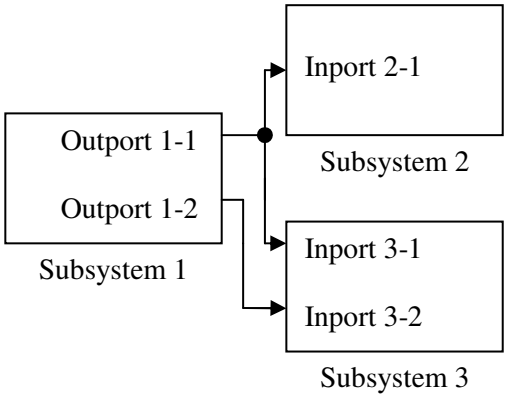
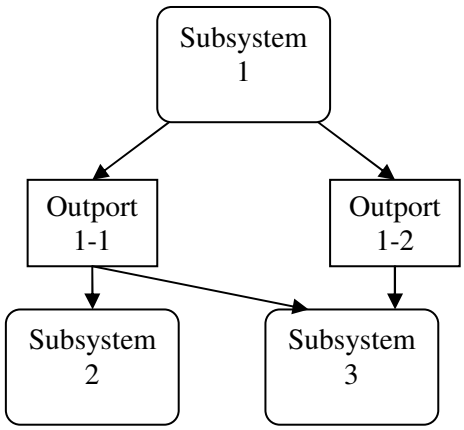
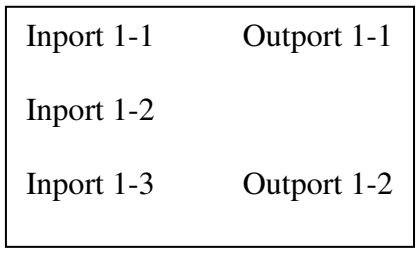
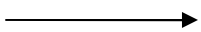
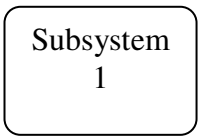
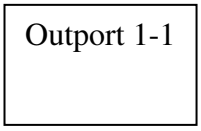
Es besteht auch die Möglichkeit, dass ein Quellport mehr als nur einen Zielport hat. Das beschreibt man im Simulinkmodell mit Hilfe von dem Schlüsselwort *Branch*, das die Verzweigungsstellen definiert. Wenn es eine Verzweigungsstelle gibt, sucht der Parser nach allen Zielblöcken, Zielports von diesen und bildet die jeweiligen Beziehungen zwischen den Knoten. In der Tabelle 4.4 ist im Detail gezeigt, wie die verschiedenen Verbindungen zwischen den Subsystemen des Simulinkmodells im Graphenmodell dargestellt werden.

Der *KLAMMER* Modus ist der einfachste Modus des Parsers. In diesem wird nur die Anzahl von geöffneten und geschlossenen geschweiften Klammern analysiert. Wenn die Differenz zwischen diesen gleich Null ist, schaltet das Programm in den *IGNORE* Modus um und sucht nach dem nächsten Block bzw. Line zum Analysieren. Dieser Modus wird von dem Parser für die Ignorierung des Inhalts der ganzen Sektionen benutzt.

Tabelle 4.4

Die Beispiele der Transformierung der verschiedenen Verbindungen zwischen den Subsystemen des Simulinkmodells mit dem MDL-Parser

Simulinkmodelldarstellung	**Graphenmodelldarstellung**
	

Simulinkmodelldarstellung	**Graphenmodelldarstellung**
	
	
Subsystem:  Subsystem 1 **Inportnamen:** Inport 1-1, Inport 1-2, Inport 1-3 **Outportnamen:** Outport 1-1, Outport 1-2 **Subsystemname:** Subsystem 1 **Line:** 	**Operationsknoten:**  **Datenknoten:**  **IDs der Knoten:** Subsystem 1, Outport 1-1

Der entwickelte Parser entspricht den technischen Bestimmungen des MDL-Dateiparsings, aber er hat die folgenden Beschränkungen.

1. Die MDL-Datei darf keine Umlaute und „ß“ enthalten, weil es dann Probleme mit dem Encoding gibt und der Prozess des Parsings abstürzt.

2. Die nonatomaren Subsysteme dürfen keine atomaren Subsysteme enthalten. Falls diese Bedingung nicht erfüllt wird, werden die inneren atomaren Subsysteme ignoriert, weil der ganze Inhalt der nonatomaren Subsysteme mit dem KLAMMER Modus ignoriert werden soll.

3. Die Subsystemnamen dürfen nur einmal vorhanden sein, weil die Suche nach den Namen der Quell- und Zielblöcke in der „Datenbank“ (Array) bei der Lineanalyse den in einem Exemplar vorhandenen Blocknamen entsprechend durchgeführt wird. Die Wahrscheinlichkeit, dass eine solche Kollision auftritt, ist sehr niedrig, weil es im Simulinkmodell verboten ist, die gleichen Namen von Subsystemen zu definieren.

4. Die Anzahl der Subsysteme ist mit der Konstanten *int grenze=1000* begrenzt. Das bedeutet, dass es im gesamten Simulinkmodell nur 1000 Subsysteme geben darf. Falls ein Modell mehr als 1000 Subsysteme enthält, muss man den Wert der Konstante im Programmtext der Anzahl der Subsysteme entsprechend korrigieren um den Auftritt des *ArrayIndexOutOfBound* Fehlers zu überwinden.

Die Schwachstelle des entwickelten MDL-Parsers besteht auch in der Aufzählung von Typen der Simulink-Komponenten (von Blocktypen) in dem Token *BLOCK_TYPE* (siehe Tabelle 4.3). In der für die Tests verwendeten Version des Parsers sind in diesem Token nur die Komponenten aufgezählt, die man für die Analyse der im Fachgebiet Rechnerarchitektur entwickelten Simulinkmodelle für die Meßtechnik braucht. Beim realen MDL-Parser, der in Simulink eingebaut ist, soll ein solches Problem durch die Einsetzung eines Modus pro Komponententyp gelöst werden.

4.6. Mögliche Fehler beim Starten der Parser

Um die entwickelten Parser der mathematischen Formeln und der MDL-Dateien für alle Benutzer für jede beliebigen mathematischen bzw. MDL-Modelle, die der in den Punkten 4.4, 4.2 erwähnten Anforderungen entsprechen, verwendet werden können, sind in der folgenden Tabelle (Tabelle 4.5) die möglichen Fehler dargestellt, die beim Starten der Parser auftreten können, und *die Quick-fixes* für diese.

Tabelle 4.5

Die möglichen auftretenden Fehler beim Starten der Parser

Anzeigebeispiel	Beschreibung des Fehlers	Quick-fix
Parser der mathematischen Formeln		
sin NOK. Encountered " "(" "("" at line *2*, column *5*. Was expecting one of: "+" ... "-" ... "*" ... "/" ... "^" ... ";" ...	Die Leerzeichen zwischen dem Funktionsnamen *sin* oder *cos* und der geöffneten Klammer (z.B. sin (x), cos (x)).	Der Benutzer muss die mathematische Formel neu angeben (ohne die Leerzeichen zwischen dem Funktionsnamen *sin* oder *cos* und der geöffneten Klammer (d.h. sin(x), cos(x)).
MDL-Parser		
NOK. Encountered " "Block {" "Block { "" at line *4412*, column *9*. Was expecting: <BLOCK_TYPE> ...	Die Simulink Komponentenbeschreibung (Blocktyp) fehlt im Token *BLOCK_TYPE* (siehe Tabelle 4.3).	Der Benutzer muss den fehlenden Blocktyp, der sich in Line *4412* des aus MDL-Datei erzeugten Textfiles befindet, zur Aufzählung im Token *BLOCK_TYPE* (durch |) hinzufügen und den Parser neu generieren.
Oops. Lexical error at line *1654*, column *47*. Encountered: "\u0446" (1094), after : ""	Die MDL-Datei enthält Umlaute oder „ß“.	Der Benutzer muss das für das Parsing falsche ASCII-Zeichen (ü, ö, ä oder ß), das sich in Line *1654* des aus MDL-Datei erzeugten Textfiles befindet, mit dem anderen (z.B. ue, oe, ae oder ss) ersetzen.
NOK. 1000	Die Anzahl der atomaren Subsysteme in der MDL-Datei ist mehr als 1000.	Der Benutzer muss den Wert der Konstante *int grenze=1000* der Anzahl der atomaren Subsysteme in der MDL-Datei entsprechend ändern und den Parser neu generieren.
Oops. Lexical error at line *5231*, column *43*. Encountered: "A" (65), after : "*?*"	Der Blockname (Simulink Komponentenname) der MDL-Datei enhält für das Parsing falsches ASCII-Zeichen (z.B. in diesem Fall "*?*").	Der Benutzer muss das für das Parsing falsche ASCII-Zeichen (siehe Token *LETTER* in der Tabelle 4.3), das sich in Line *5231* des aus MDL-Datei erzeugten Textfiles befindet, mit dem anderen ersetzen oder löschen.

4.7. Modellkonverter und Optimierer

Die Anforderungen an die Funktionalität des Modellkonverters wurden schon im Punkt 3.3 erläutert. Im Folgenden wird Schritt für Schritt der Algorithmus des Konverters vorgestellt.

Die wichtigste Funktion des Modellkonverters ist die Partitionierung (Mapping) des Eingangsmodells, das die Graph- und Plattformbeschreibung enthält, auf die Ressourcen (CPUs und Busse) dieser Plattform. Um das zu realisieren, soll der Konverter das Verhalten des realen Rechnersystems modellieren, als ob auf diesem der gegebene Graph als Task ausgeführt wird. Dafür braucht man die folgenden Datenstrukturen im Programmcode des Konverters zu definieren.

a) Für die qualitative und quantitative Analyse des Graphenmodells:

- *Integer Array* $t_{min}[N]$ für die Berechnung von $T_{min\ i}$ der Formel (3.1) entsprechend für jeden Operationsknoten; *N* ist die Anzahl der Operationsknoten des Graphenmodells;

- *Integer Array* $t_{max}[N]$ für die Berechnung von $T_{max\ i}$ der Formel (3.2) entsprechend für jeden Operationsknoten;

- *Integer* T_{max} für die Berechnung der Ausführungszeit des Graphen auf dem Rechnersystem, der nur aus einem Prozessor besteht (sequentielles Rechnersystem);

- *Integer* T_{min} für die Berechnung der Ausführungszeit des Graphen auf dem Rechnersystem, mit unbegrenzten Ressourcen;

- *Integer Array root[N]* für die Speicherung der Identifikationsnummern (*IN*s) der Operationsknoten, die die Wurzelknoten des Graphen sind. *IN* ist die Reihenfolgenummer des Knotens im Graphenmodell (1..*N*) bzw. Prozessor im Plattformmodell (1..*M*), die automatisch bei der Erzeugung des Knotens (der Ressource) generiert wird.

b) Für das Mapping

- *Integer Array master[M]* für die Speicherung der INs der Masterprozessoren; *M* ist die Anzahl der Prozessoren der Plattform;

- *Integer strategie* für die Auswahl der Ausführungsstrategie;

- *Integer on_spot* für das Ein- und Ausschalten des *On-spot Mapping* Modus (wird erklärt später);

- *Integer t* entspricht der aktuellen Modellzeit;

- *Integer Matrix cpu_busy[M][3]* entspricht dem Status der Prozessoren (3 Elemente pro Prozessor der Plattform):

  - Falls *cpu_busy[i][0] gleich* 0 ist, ist Prozessor # *i* zur Zeit frei; sonst ist er besetzt.
  - *cpu_busy[i][1]* entspricht dem Modellzeitpunkt, ab dem Prozessor # *i* besetzt wird.
  - *cpu_busy[i][2]* entspricht dem Modellzeitpunkt, ab dem Prozessor # *i* wieder frei ist.

- *Integer Matrix node_status[N][4]* entspricht dem Status der Operationsknoten (4 Elemente pro Operationsknoten des Graphen):

  - Die möglichen Werte der Variablen *node_status[j][0]* sind in der Tabelle 4.6 dargestellt.

Tabelle 4.6

Die möglichen Status der Operationsknoten des Graphen

Status (node_status[j][0] =)	**Bedeutung**
-1	Der Operationsknoten # *j* kann noch nicht gemappt (gestartet) werden, weil nicht alle seine Vorgänger ausgeführt sind.
0	Der Operationsknoten # *j* kann gemappt (gestartet) werden.
1	Der Operationsknoten # *j* ist gemappt und gestartet, aber noch nicht bis zum Ende ausgeführt.
2	Der Operationsknoten # *j* ist gemappt und bis zum Ende ausgeführt.

  - *node_status[j][1]* entspricht dem Modellzeitpunkt, ab dem der Operationsknoten # *j* ausgeführt wird.
  - *node_status[j][2]* entspricht dem Modellzeitpunkt, wenn der Operationsknoten # *j* ausgeführt ist.
  - *node_status[j][3]* entspricht der Zeit, während der der Operationsknoten # *j* sich in der Warteschlange (siehe *integer map_queue*) befunden hat.

- *Integer Matrix map_queue[N][3]* stellt die Warteschlange der Knoten, die gemappt werden müssen, vor (3 Elemente pro Knoten in der Warteschlange):

- *map_queue[][0]* sind die INs der Operationsknoten (der Reihenfolge des Graphen gemäß), die sich zur Zeit in der Warteschlange befinden.
- *map_queue[][1]* präsentieren die Ausführungszeiten von diesen Operationsknoten.
- Falls *map_queue[k][2] gleich* 0 ist, gehört der Operationsknoten # k der Warteschlange zum kritischen Pfad des Graphen, sonst nicht.

Wie man aus der Beschreibung der oben erwähnten Datenstrukturen verstehen kann, wird der entwickelte Algorithmus des Modellkonverters keine Übertragungszeiten berücksichtigen. Aus diesem Grund werden im Folgenden (in diesem Punkt) die Operationsknoten einfach als Knoten genannt und als Knoten der parallelen Schichtform betrachtet, weil die Datenknoten, die mit den Übertragungszeiten verbunden sind, im entwickelten Algorithmus nicht berücksichtigt werden, d.h. alle $t_{ü} = 0$.

Die graphische Darstellung des Algorithmus der Modellkonvertierung ist in Abb. 4.4 dargestellt und im Folgenden ist seine detaillierte schrittweise Beschreibung angegeben.

1. Das Graphen- und das Plattformmodell (Eingangsmodell) werden eingelesen.

2. Die Wurzelknoten des Graphen werden gesucht und die INs von diesen werden aus dem Graphenmodell in das Array *root[]* gespeichert. Die Wurzelknoten sind die Operationsknoten, für die die *Argumente* keine *Producers* haben (siehe *A1* und *S1* in Abb. 4.2) oder die überhaupt keine *Argumente* haben (Simulink-Komponenten nur mit Outports).

3. $T_{min\ i}$ der Wurzelknoten des Graphen wird auf *0* gesetzt und $T_{min\ i}$ für alle anderen Knoten werden der Formel (3.1) gemäß berechnet und im Array $t_{min}[]$ den INs entsprechend gespeichert.

4. Die Blätter des Graphen werden gesucht und $T_{max\ i}$ von diesen werden auf jeweiligen $T_{min\ i}$ gesetzt. Die Blätter sind die Operationsknoten, für die die *Results* keine *Consumers* haben (siehe *M1* in Abb. 4.2 und Abb. 3.4) oder die überhaupt keine *Results* haben (Simulink-Komponenten nur mit Inports). $T_{max\ i}$ für alle anderen Knoten werden der Formel (3.2) gemäß berechnet und im Array $t_{max}[]$ den INs entsprechend gespeichert.

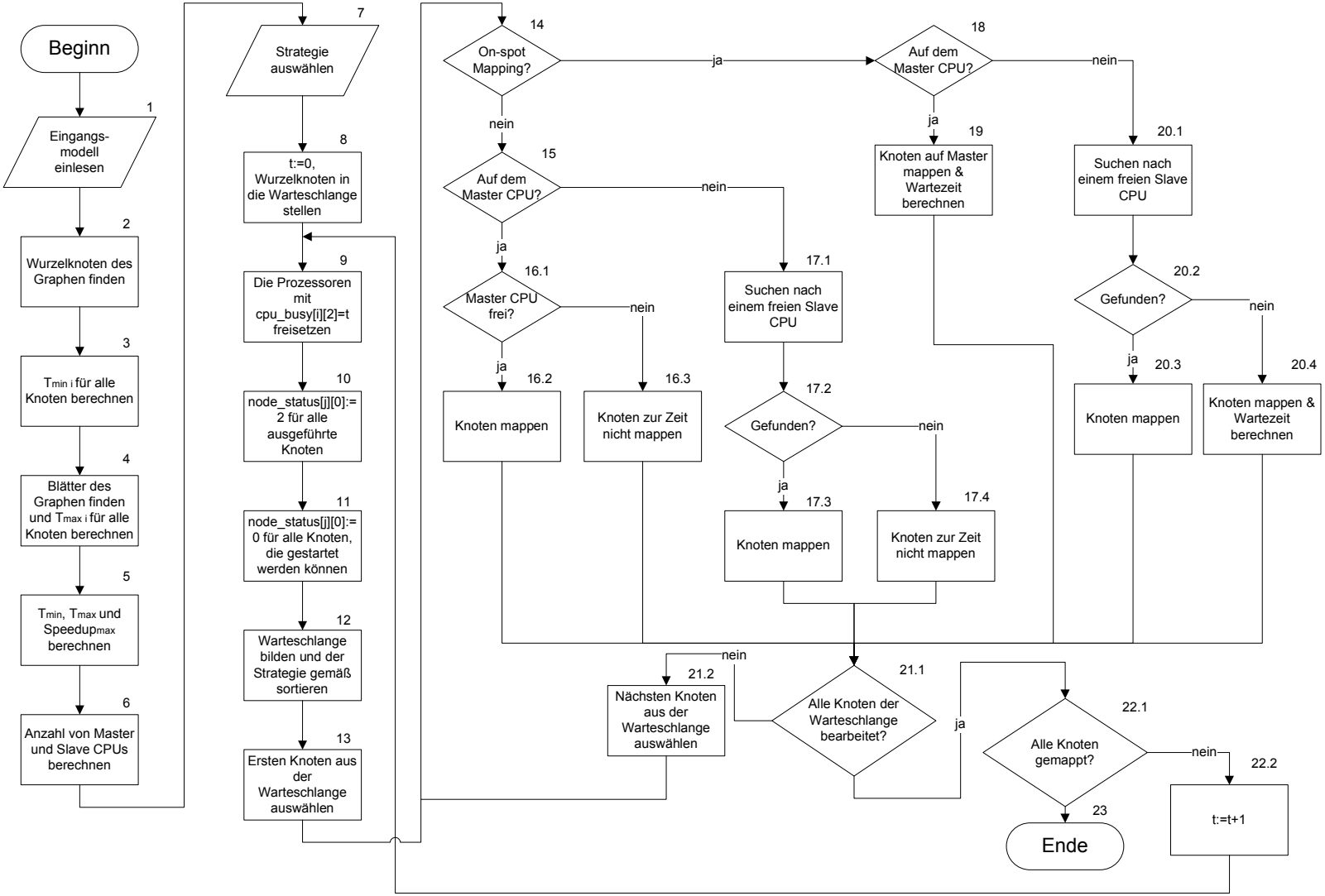


Abb. 4.4. Algorithmus der Modellkonvertierung

5. T_{min}, T_{max} von dem ganzen Graphen und $Speedup_{max}$ (3.3) werden berechnet. $T_{min} = (max_{i=1..N}\ T_{min\ i}) + t_{a\ i}$ und $T_{max} = \sum_{i=1}^{N} t_{a\ i.}$ Der kritische Pfad des Graphen enthält die Knoten aus jeder Schicht (1 Knoten pro Schicht, siehe Abb. 3.7), die im Modell miteinander verbunden sind und für die $T_{max\ i} = T_{min\ i}$.

6. Die Anzahl von *Master* und *Slave* Prozessoren wird aus dem Plattformmodell eingelesen und die INs der *Master*s werden im Array *master[]* gespeichert.

7. Die Ausführungsstrategie wird vom Nutzer ausgewählt. Insgesamt werden 4 Strategien vorgeschlagen, das sind Strategien „random“, „minimal“, „maximal“ und „kritischer Pfad“. Sie unterscheiden sich im Prinzip, nach dem die Warteschlange der Knoten (*map_queue[][]*), die gemappt werden müssen, vor dem Mapping sortiert wird (siehe Tabelle 4.7).

Tabelle 4.7

Ausführungsstrategien

Strategie	**Sortierungsprinzip**	**Beispiel (*map_queue[][]*)**	
		Vor der Sort.	**Nach der Sort.**
Random	Die Knoten werden den INs gemäß (zufällig) sortiert.	{2, 10, 0} {1, 20, 1} {8, 60, 1}	{**1**, 20, 1} {**2**, 10, 0} {**8**, 60, 1}
Minimal	Die Knoten, die minimale Ausführungszeit haben, werden zuerst gemappt.	{2, **10**, 0} {1, **20**, 1} {8, **60**, 1}	{2, **10**, 0} {1, **20**, 1} {8, **60**, 1}
Maximal	Die Knoten, die maximale Ausführungszeit haben, werden zuerst gemappt.	{2, **10**, 0} {1, **20**, 1} {8, **60**, 1}	{8, **60**, 1} {1, **20**, 1} {2, **10**, 0}
Kritischer Pfad	Die Knoten, die zum kritischen Pfad gehören, werden zuerst gemappt.	{2, 10, 0} {1, 20, 1} {8, 60, 1}	{2, 10, **0**} {1, 20, **1**} {8, 60, **1**}

Falls es in der Warteschlange mehrere Knoten mit der gleichen Ausführungszeit gibt, werden sie innerhalb der Ausführungszeiten den INs gemäß sortiert.

Es besteht auch die Möglichkeit jede Strategie mit oder ohne *on-spot Mapping* durchzuführen. Beim *on-spot Mapping* werden alle Knoten, die sich in der Warteschlange befinden, sofort gemappt, unabhängig davon, wie viele freie Prozessoren es zur Zeit gibt. Sonst, wenn es keine freien Prozessoren gibt, müssen die Knoten, die zur Zeit nicht gemappt werden können aber in der Warteschlange sind, weiter die in der Warteschlange warten.

8. Die Modellzeit *t* wird auf *0* gesetzt. Die Wurzelknoten werden in die Warteschlange gestellt.

9. Falls es die Prozessoren gibt, die in diesem Zeitpunkt freigesetzt werden können (für die *cpu_busy[i][2]* = t), müssen diese freigemacht werden (*cpu_busy[i][0]* := 0).

10. Falls es die Knoten gibt, die in diesem Zeitpunkt bis zum Ende ausgeführt sind (für die *node_status[j][2]* = t), muss für jeden von diesen *node_status[j][0]* auf *2* gesetzt werden.

11. Es wird gesucht nach den Knoten, die in diesem Zeitpunkt gestartet werden können. Das sind die Knoten, für die alle Vorgänger ausgeführt sind und *node_status[j][0]* = -1. Falls es solche Knoten gibt, wird für jeden von diesen *node_status[j][0]* auf *0* gesetzt.

12. Alle Knoten mit *node_status[j][0] = 0* werden in die Warteschlange hinzugefügt. Die Warteschlange wird der Ausführungsstrategie entsprechend sortiert.

13. Der erste Knoten aus der Warteschlange wird für die Bearbeitung ausgewählt (wird als der aktuelle Knoten bestimmt).

14. Falls *on-spot Mapping* aktiviert ist, dann zum Schritt 18.

15. Es wird geprüft, ob der aktuelle Knoten (mit IN = *j*) auf dem *Master* CPU ausgeführt werden müssen. Falls nein, dann zum Schritt 17.

16. Falls Master CPU zur Zeit frei ist, wird der aktuelle Knoten auf den *Master* gemappt und es werden die folgenden Werte der Datenstrukturen gesetzt:

- cpu_busy[master IN][0] := 1 (*Master* besetzen);
- cpu_busy[master IN][1] := t;
- cpu_busy[master IN][2] := t + $t_{a\,j}$;
- node_status[j][0] := 1;
- node_status[j][1] := t;
- node_status[j][2] := t + $t_{a\,j}$;

Sonst, wird der aktuelle Knoten zur Zeit nicht gemappt (node_status[j][3] ++).

Weiter zum Schritt 21.

17. Es wird nach dem ersten (IN gemäß) freien *Slave* CPU gesucht. Wenn es einen freien *Slave* gibt, wird der aktuelle Knoten auf diesen gemappt und es werden die folgenden Werte der Datenstrukturen gesetzt:

- cpu_busy[slave IN][0] := 1 (*Slave* besetzen);
- cpu_busy[slave IN][1] := t;

- cpu_busy[slave IN][2] := t + $t_{a\,j}$;
- node_status[j][0] := 1;
- node_status[j][1] := t;
- node_status[j][2] := t + $t_{a\,j}$;

Sonst, wird der aktuelle Knoten zur Zeit nicht gemappt (node_status[j][3] ++).

Weiter zum Schritt 21.

18. *On-spot Mapping*. Es wird geprüft, ob der aktuelle Knoten (mit IN= *j*) auf dem *Master* CPU ausgeführt werden muss. Falls nein, dann zum Schritt 20.

19. Der aktuelle Knoten wird auf den *Master* gemappt. Falls Master CPU zur Zeit frei ist, werden die folgenden Werte der Datenstrukturen gesetzt:

- cpu_busy[master IN][0] := 1 (*Master* besetzen);
- cpu_busy[master IN][1] := t ;
- cpu_busy[master IN][2] := t + $t_{a\,j}$;
- node_status[j][0] := 1;
- node_status[j][1] := t;
- node_status[j][2] := t + $t_{a\,j}$;
- node_status[j][3] := 0;

Sonst die folgenden:

- node_status[j][0] := 1;
- node_status[j][1] := cpu_busy[master IN][2] (starten sobald der *Master* frei ist);
- node_status[j][2] := cpu_busy[master IN][2] + $t_{a\,j}$;
- node_status[j][3] := cpu_busy[master IN][2] - t;
- cpu_busy[master IN][2] := cpu_busy[master IN][2] + $t_{a\,j}$;

Weiter zum Schritt 21.

20. Es wird nach dem ersten (IN gemäß) freien *Slave* CPU gesucht. Wenn es einen freien *Slave* gibt, wird der aktuelle Knoten auf diesen gemappt und es werden die folgenden Werte der Datenstrukturen gesetzt:

- cpu_busy[slave IN][0] := 1 (*Slave* besetzen);
- cpu_busy[slave IN][1] := t ;
- cpu_busy[slave IN][2] := t + $t_{a\,j}$;
- node_status[j][0] := 1;

- node_status[j][1] := t;
- node_status[j][2] := t + $t_{a\,j}$;
- node_status[j][3] := 0;

Sonst wird der aktuelle Knoten auf den *Slave* mit der mindesten cpu_busy[slave IN][2] Zeit gemappt und es werden die folgenden Werte der Datenstrukturen gesetzt:

- node_status[j][0] := 1;
- node_status[j][1] := cpu_busy[slave IN][2] (starten sobald der *Slave* frei ist);
- node_status[j][2] := cpu_busy[slave IN][2] + $t_{a\,j}$;
- node_status[j][3] := cpu_busy[slave IN][2] - t;
- cpu_busy[slave IN][2] := cpu_busy[slave IN][2] + $t_{a\,j}$;

21. Falls noch nicht alle Knoten der Warteschlange bearbeitet sind, wird der nächste Knoten ausgewählt und der Algorithmus vom Schritt 14 wiederholt.

22. Falls noch nicht alle Knoten des Graphen gemappt sind, wird die Modellzeit *t* inkrementiert (*t*++) und der Algorithmus vom Schritt 9 wiederholt.

23. Der Algorithmus des Mappings wird beendet und das mit dem Mapping erweiterte Eingangsmodell wird im *.archmap Format gespeichert.

Das Eingangsmodell sowie das erweiterte Modell, die im *.archmap Format gespeichert sind, können weiter im *Eclipse Editor* geöffnet werden (Abb. 4.5). Das ganze Modell wird in Form einer Baumstruktur dargestellt (Abb. 4.5, *links*) und der Nutzer kann damit die Eigenschaften jedes Modellelements (Knotens des Graphen, Ressource der Plattform, Mappings) anschauen und bei Bedarf ändern (Abb. 4.5, *rechts*).

Bei der Ausführung des entwickelten Algorithmus auf dem Rechnersystem, das dem von Benutzer angegebenen Plattformmodell entspricht, können die berechneten Werte der Parameter des Modells (T_{min}, $Speedup_{max}$ u.a.) sehr weit von den realen gemessenen Werten liegen, weil das Modell selbst sehr vereinfacht ist. Im Punkt 4.8 wird gezeigt, wie man das Plattformmodell erweitern kann, um diese Ungenauigkeit zu vermindern.

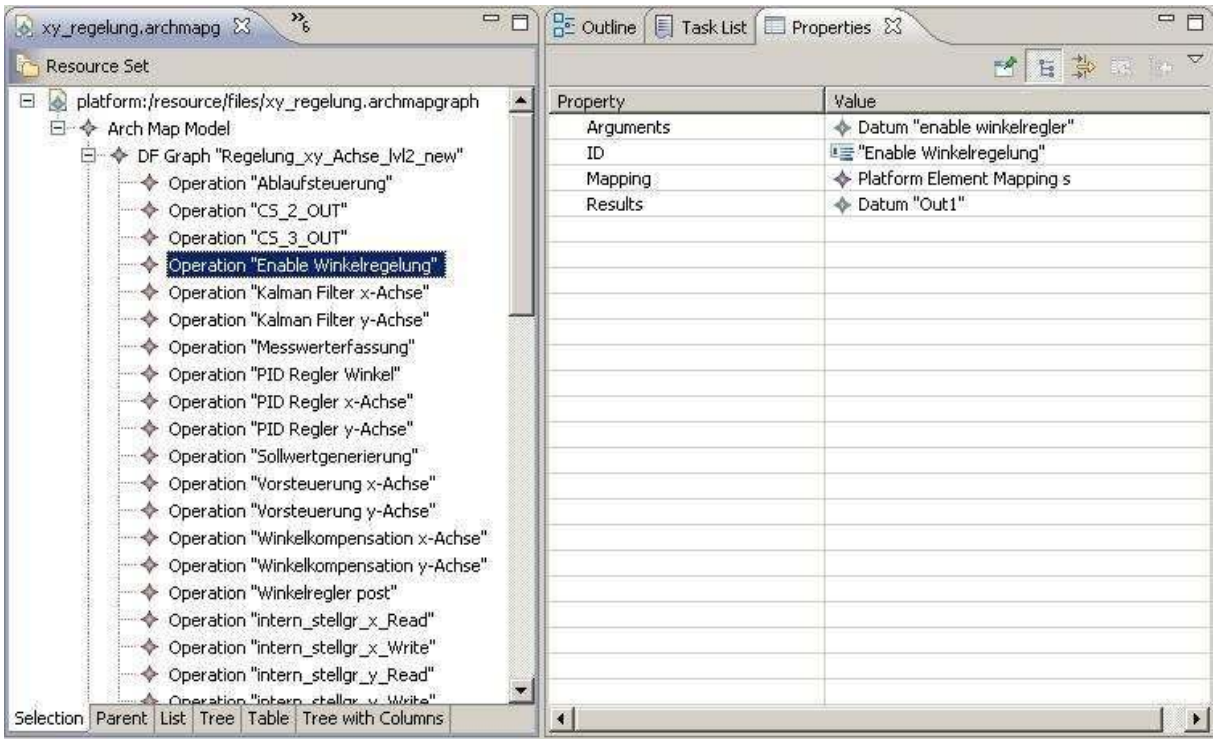


Abb. 4.5. Eclipse-Editor für Datenmodelle

4.8. Berücksichtigung der Übertragungszeiten bei der Modellkonvertierung

Im entwickelten Algorithmus der Modellkonvertierung werden die Übertragungszeiten der Daten zwischen den Prozessoren des Rechnersystems nicht berücksichtigt. Dieser Fakt kann große Ungenauigkeiten bei der Bewertung der Parameter des Modells auslösen, falls der Graph, der ausgeführt werden muss, mittel oder stark gebunden ist (siehe Punkt 3.3.1) oder die Kommunikationsmechanismen des Rechnersystems selbst sehr langsam sind.

Die Rechnersysteme können sich in der Architektur, der Speicherhierarchie und den Kommunikationsverfahren zwischen den Prozessoren voneinander unterscheiden. Diese Eigenschaften des Rechnersystems üben verschiedene Einwirkungen auf die Berechnung der Datenübertragungszeiten aus und führen dazu, dass die Übertragungszeiten zwischen den verschiedenen Prozessoren des Rechnersystems für die gleichen Datengrößen nicht identisch sind.

In Folgendem wird für jeden Typ des Rechnersystems mit *dem verteilten Speicher* gezeigt, wie man die Datenübertragungszeiten berechnen kann. Solche Rechnersysteme entsprechen am meisten dem im Punkt 2.2 beschriebenen *MDSP-System* und sind nur die Modifikationen davon. Dafür wird das auf dem Bewertungskoeffizienten basierende Verfahren verwendet, mit Hilfe von diesem man beim Multiplizieren der mittleren Übertragungszeit (n / V_{BUS}) mit dem jeweiligen Koeffizient (4.2) die Datenübertragungszeit $t_{Üi}$ berechnen kann:

$$t_{Ü\,i} = K^* (n / B_{BUS}{*}V_{BUS}) \qquad (4.2).$$

Dabei entspricht K dem Bewertungskoeffizient, n der Datengröße, V_{BUS} der Geschwindigkeit und B_{BUS} der Breite des für die Übertragung verwendeten Busses.

4.8.1. Rechnersysteme mit dem verteilten Speicher

Die Rechnersysteme mit dem verteilten Speicher (*distributed memory*) bestehen aus M Prozessoren $P_1...P_M$, $M \geq 2$ und jeder von diesen enthält seinen eigenen lokalen Speicher (LM_i), das Input/Output (I/O) Teilsystem und das Betriebssystem (oder Teil des Betriebssystems). Die Prozessoren $P_1...P_M$ werden auch als Rechenknoten (RK) definiert (Abb. 4.6), weil sie eine Menge von voneinander unabhängigen Rechnern vorstellen, und deshalb wird das gesamte Rechnersystem als das *Multirechnersystem* genannt.

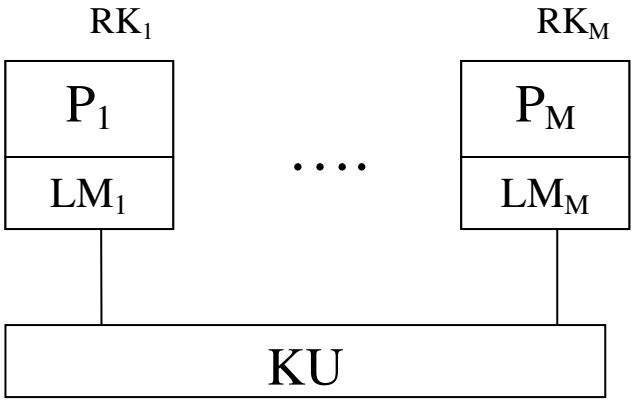


Abb. 4.6. Rechnersystem mit dem verteilten Speicher

Die Kommunikationsumgebung (KU) zwischen den Prozessoren enthält typischerweise von *1* bis zu *M* Busse, die mit jedem CPU verbunden sind. Solche Konfiguration der Ressourcen des Rechnersystems entspricht am meisten dem im Punkt 3.2.2.2 dargestellten Plattformmodell.

Die grundlegende Formel für die Berechnung der durchschnittlichen Datenübertragungszeit zwischen zwei Prozessoren des Rechnersystems mit dem verteilten Speicher ist folgende (4.3):

$$t_{Ü\,i} = K_{ÜTYP} * K_{Verz} (n / B_{BUS} * V_{BUS}) \qquad (4.3).$$

Der Bewertungskoeffizient $K_{ÜTYP} = 0$, falls die Datenübertragung zwischen CPU_i und seinen eigenen lokalen Speicher LM_i ausgeführt wird. Wenn die Daten zwischen zwei verschiedenen Prozessoren übertragen werden müssen, dann $K_{ÜTYP} = 1$, und das berücksichtigt die Zeit, die für den einseitigen Datenaustausch zwischen zwei LMs gebraucht wird. $K_{Verz} \geq 1$ ist der Verzögerungskoeffizient, der von der Organisation des Rechnersystems abhängig ist und vom Nutzer heuristisch der Ursache der Verzögerung gemäß (siehe Tabelle 4.8) bewertet werden muss.

Tabelle 4.8

Ursachen der Verzögerungen bei der Datenübertragung für die Rechnersysteme mit dem verteilten Speicher

Beschreibung des Rechnersystems	Verzögerungen
Rechnersystem mit M Rechenknoten und 1 Bus. Jeder Knoten ist mit dem Bus verbunden. Die Knoten funktionieren dem *Master-Slave* Prinzip entsprechend (siehe die Beschreibung des *MDSP-System*s im Punkt 2.2).	Der Prozess der Datenübertragung wird immer von *Master* ausgeführt. Die Zeit der Verzögerung ist immer (ungefähr) gleich und hängt von der Geschwindigkeit des Masters ab.
Rechnersystem mit M Rechenknoten und 1 Bus. Jeder Knoten ist mit dem Bus verbunden. Es gibt keine *Master*-Knoten, jeder Knoten ist absolut selbständig.	Der gemeinsame Bus ist ein „bottleneck" solches Systems, der zu den Verzögerungen führt. Die speziellen Mechanismen müssen auch entwickelt werden, um die Kollisionen (gleichzeitige Datenübertragungen) zu vermeiden.
Rechnersystem mit M Rechenknoten und M Busse. Jeder Knoten ist mit jedem Bus verbunden (insgesamt M*M Verbindungen).	Die Verzögerung hängt davon ab, wie viele Anfragen der lokale Speicher bearbeiten kann. Falls der lokale Speicher bis zu M Anfragen bearbeiten kann, gibt es keine Verzögerungen.

Kapitel 5

Zusammenfassung und Ausblick

Im Verlauf dieser Masterarbeit ist eine Javaaplikation entworfen worden, die allen in der Einleitung vorgestellten Zielen entspricht. Diese Applikation stellt dem Nutzer den Mechanismus für die Transformierung von 2 Arten von Eingangsdatenformaten, das sind die mathematischen Formeln und die Dateien des MatLab-Simulinkmodells (MDL), in den Datenflussgraphen zur Verfügung. Die erzeugten Graphen werden weiter auf die Architektur eines Zielrechnersystems, die vom Nutzer in Form der *CPU-Bus* Verbindungen vorgegeben wird, partitioniert.

Die Datenflussgraphen enthalten 2 Arten von Knoten, das sind Operationsknoten und Datenknoten. Jeder Knoten hat seine eigene ID, die für jede Art von Eingangsdaten eine andere Bedeutung hat. Die Operationsknoten des Graphen werden auf die Rechnerressourcen (Prozessoren) des Zielrechnersystems partitioniert und die Datenknoten auf die Kommunikationsressourcen (Busse).

Die Umwandlung der Eingangsdaten in den Datenflussgraphen wird mit dem speziellen Programm, dem Parser, ausgeführt, der der in dieser Arbeit entwickelten Grammatikregeln (Parsergrammatik) entsprechend mit Hilfe von JavaCC Plug-in für Eclipse generiert wird. Die Partitionierung des Graphen auf die Ressourcen des Zielrechnersystems erfolgt ihrerseits dem entwickelten und im Punkt 4.7 schrittweise beschriebenen Algorithmus gemäß. Dieser Partitionierungsalgorithmus funktioniert in einem von 2 Modi und nach einer von 5 Ausführungsstrategien, die vom Nutzer ausgewählt werden. Die Modellzeit im Algorithmus ist diskret mit dem Einheitsschritt (0, 1, 2 ...). Die Zeiteinheiten können als Takte der Prozessoren oder als ms (µs usw.) interpretiert werden.

Während der Ausführung des Partitionierungsalgorithmus wird die qualitative und quantitative Datenflussgraphenanalyse durchgeführt und solche wichtigen Parameter wie die minimal mögliche Ausführungszeit des Graphen auf dem Rechnersystem mit

unbegrenzten Ressourcen (T_{min}) und der maximale Speedup ($Speedup_{max}$) berechnet, damit man entweder die optimale (minimale) Ausführungszeit des Graphen auf dem vorgegebenen Rechnersystem bewerten oder das optimale Rechnersystem, auf dem der Graph in T_{min} ausgeführt werden kann, durch die Variierung der CPU- und Busanzahl im Modell der Zielrechnersystemarchitektur finden kann.

Für die Eingangsdatenparser sind im Text der Arbeit die möglichen auftretenden Fehler und die Quick-fixes für diese vorgestellt und im Anhang A sind auch die Tests der Funktionalität der MDL-Parser an einem MDL-Dateibeispiel für ein komplexes Reglersystem präsentiert.

Im Rahmen dieser Arbeit sind die Schwerpunkte sowie die Schwachstellen für die entwickelten Parser und Partitionierungsalgorithmus erläutert, damit man objektiv verstehen kann, welche Bestandteile der Javaaplikation noch implementiert oder verändert werden können.

Das Plattformmodell *Platform* kann in der Weiterentwicklung des Metamodells durch die Speicherhierarchie des Zielrechnersystems ergänzt werden.

Der Partitionierungsalgorithmus berücksichtigt bis jetzt keine Datenübertragungszeiten zwischen den Prozessoren des Zielrechnersystems. Das kann die großen Ungenauigkeiten bei der Bewertung der Parameter des Datenflussgraphen auslösen, deshalb ist die Realisierung des im Punkt 4.8 beschriebenen Konzepts für die Berücksichtigung der Übertragungszeiten ein vielversprechender Ansatz zur Erweiterung der entwickelten Applikation.

Der Partitionierungsalgorithmus kann auch so erweitert werden, dass man damit die gleichzeitige parallele Ausführung von 2 oder mehreren Datenflussgraphen auf dem Zielrechnersystem modellieren kann.

Anhang A

Funktionale Tests des MDL-Parsers

In diesem Anhang werden die funktionalen Tests des MDL-Parsers an dem MDL-Dateibeispiel *Regelung_xy_Achse_lvl2_new.mdl* für ein komplexes Reglersystem (Abb. A.1) vorgestellt, das im Sonderforschungsbereich 622 „Nanopositionier- und Nanomessmaschinen“ der TU Ilmenau entwickelt wird.

Die oben erwähnte MDL-Datei wurde mit dem im Punkt 4.5 beschriebenen Parser bearbeitet und in den Datenflussgraphen transformiert. Als nächster Schritt wurde dieser Datenflussgraph mit dem Modellkonverter *ArchMappingConverter* in Form einer *.archmap-Datei (*xy_regelung.archmapgraph*) eingelesen und auf dem MDSP-Zielrechnersystem (Abb. 2.2), das aus einem Master und 4 Slave Prozessoren (Slave 0 … Slave 3) besteht, partitioniert. Die Partitionierung wurde in 2 Modi und unter Nutzung von 3 Optimierungsstrategien (Tabelle 4.7) durchgeführt, um die minimale Ausführungszeit des Graphen auf dem vorgegebenen Rechnersystem zu finden. Dazu wurden auch die Graphenanalyseverfahren verwendet, damit die Bewertungsparameter, wie $T_{min\ i}$, $T_{max\ i}$, den Formeln (3.1), (3.2) entsprechend für jeden Knoten berechnet wurden.

Die Ergebnisse der Partitionierung sind in der Tabelle A.1 dargestellt.

Dem im Punkt 4.8 schrittweise beschriebenen Algorithmus gemäß wurden auch die folgenden Parameter des gesamten Graphenmodells und Mappings bewertet:

1) Kritische Pfade des Datenflussgraphen: 6-11-4-20-13-1 *oder* 6-11-4-20-13-2
6-12-5-21-14-1 *oder* 6-12-5-21-14-2

Dabei entsprechen die Zahlen den INs der Knoten des Datenflussgraphen (siehe Tabelle A.1). Alle kritischen Pfade sind in Abb. A.2 rot markiert.

2) Die minimal mögliche Ausführungszeit des Graphen auf dem Rechnersystem mit unbegrenzten Ressourcen $T_{min} = 90030$ ns.

3) Die Ausführungszeit des Datenflussgraphen auf dem sequentiellen Rechnersystem $T_{max} = 167106$ ns.

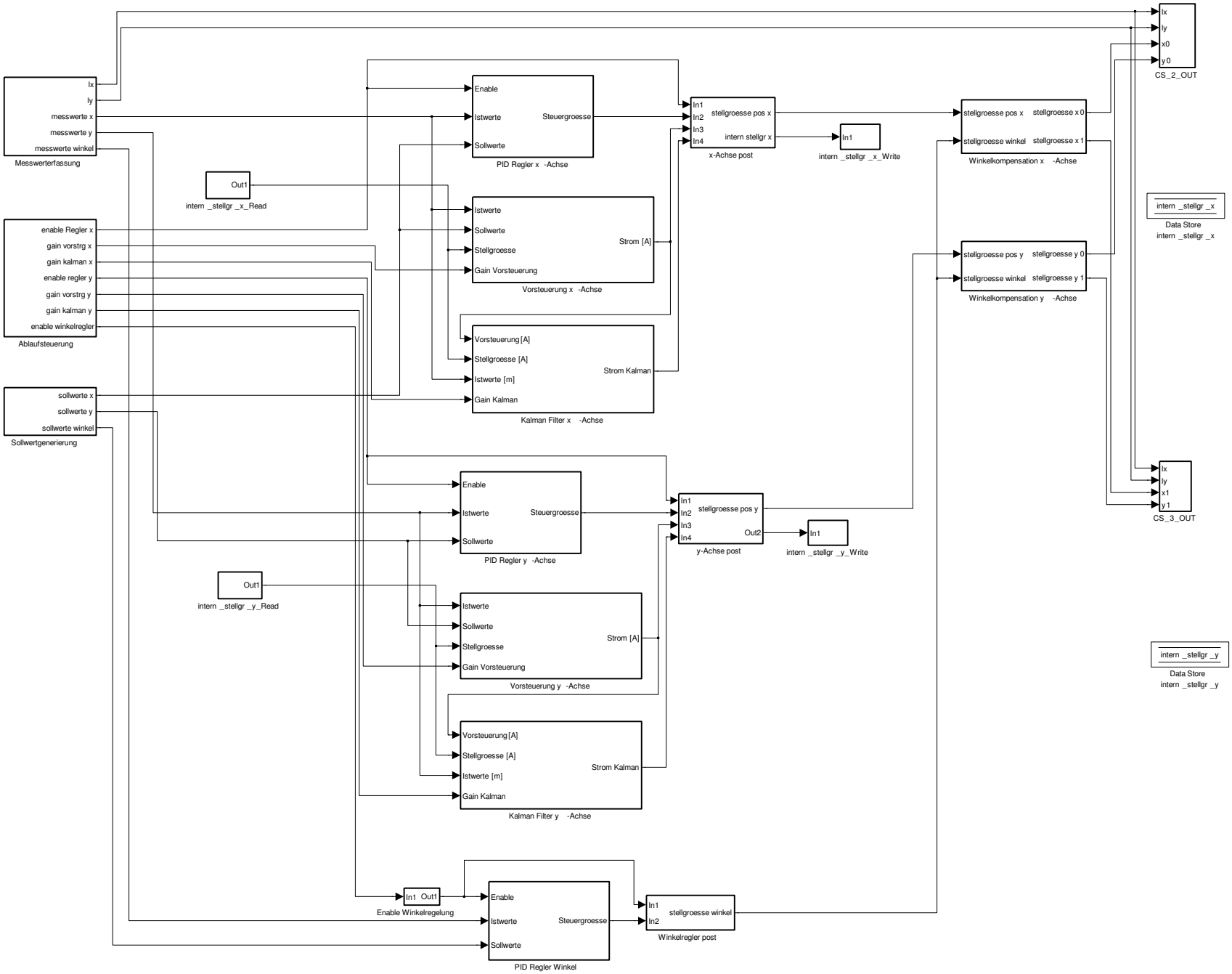


Abb. A.1. Komplexes Reglersystem (Simulink)

Tabelle A.1

Die Ergebnisse der Partitionierung des Datenflussgraphen für das komplexe Reglersystem (Abb. A.1)

IN des Knotens	ID des Knotens	$t_{a\,i}$, ns	$T_{min\,i}$, ns	$T_{max\,i}$, ns	Mit on-spot Mapping			Ohne on-spot Mapping		
					Min	Max	Kr.Pf.	Min	Max	Kr.Pf.
0	Ablaufsteuerung	1000	0	29000	Master	Master	Master	Master	Master	Master
1	CS_2_OUT	20	90010	90010	Master	Master	Master	Master	Master	Master
2	CS_3_OUT	20	90010	90010	Master	Master	Master	Master	Master	Master
3	Enable Winkelregelung	1	1000	85000	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0
4	Kalman Filter x-Achse	25000	65000	65000	Slave 1	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0
5	Kalman Filter y-Achse	25000	65000	65000	Slave 0	Slave 1	Slave 1	Slave 1	Slave 1	Slave 1
6	Messwerterfassung	30000	0	0	Master	Master	Master	Master	Master	Master
7	PID Regler Winkel	5000	30000	85000	Slave 0	Slave 2	Slave 2	Slave 0	Slave 2	Slave 2
8	PID Regler x-Achse	5000	30000	85000	Slave 1	Slave 3	Slave 3	Slave 1	Slave 3	Slave 3
9	PID Regler y-Achse	5000	30000	85000	Slave 2	Slave 2	Slave 2	Slave 2	Slave 2	Slave 2
10	Sollwertgenerierung	1000	0	29000	Master	Master	Master	Master	Master	Master
11	Vorsteuerung x-Achse	35000	30000	30000	Slave 3	Slave 0	Slave 0	Slave 3	Slave 0	Slave 0
12	Vorsteuerung y-Achse	35000	30000	30000	Slave 0	Slave 1	Slave 1	Slave 1	Slave 1	Slave 1
13	Winkelkompensation x-Achse	5	90005	90005	Slave 1	Slave 2	Slave 0	Slave 0	Slave 2	Slave 0
14	Winkelkompensation y-Achse	5	90005	90005	Slave 0	Slave 3	Slave 1	Slave 0	Slave 3	Slave 1
15	Winkelregler post	5	35000	90000	Slave 1	Slave 3	Slave 3	Slave 0	Slave 3	Slave 3
16	intern_stellgr_x_Read	10	0	29990	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0
17	intern_stellgr_x_Write	10	90005	90005	Slave 2	Slave 0	Slave 2	Slave 2	Slave 0	Slave 2
18	intern_stellgr_y_Read	10	0	29990	Slave 1	Slave 1	Slave 1	Slave 1	Slave 1	Slave 1
19	intern_stellgr_y_Write	10	90005	90005	Slave 1	Slave 1	Slave 3	Slave 1	Slave 1	Slave 3
20	x-Achse post	5	90000	90000	Slave 1	Slave 0	Slave 0	Slave 0	Slave 0	Slave 0
21	y-Achse post	5	90000	90000	Slave 0	Slave 1	Slave 1	Slave 0	Slave 1	Slave 1

4) Der maximale Speedup, den man für den gegebenen Graphen mit jedem möglichen Rechnersystem erreichen kann, $Speedup_{max} = 1{,}86$.

Für jedes in der Tabelle A.1 dargestellte Mapping wird die Ausführungszeit des Graphen auf dem MDSP-System T_a, die gesamte Wartezeit der Knoten auf die freien Prozessoren $T_{wait\ \sum}$ sowie die Auslastung der Prozessoren den unten angegebenen Formeln gemäß berechnet (Tabelle A.2).

$$T_{wait\ \sum} = \sum_{j=1}^{N} \text{node\_status}[j][3] \qquad \text{(A.1)};$$

$$\text{Auslastung der Prozessor} = T_{unter\_last} / T_a \qquad \text{(A.2)}.$$

Dabei entspricht T_{unter_last} der Zeit, während der der jeweilige Prozessor mit der Berechnung der Operationsknoten des Graphen besetzt ist.

Tabelle A.2

Die quantitativen Bewertungen der Partitionierung des Datenflussgraphen für das komplexe Reglersystem (Abb. A.1)

Strategie	**T_a, ns**	**$T_{wait\ \sum}$, ns**	**Die Auslastung der Prozessoren, %**				
			Slave 0	**Slave 1**	**Slave 2**	**Slave 3**	**Master**
Mit on-spot Mapping							
Min	97050	8020	67,0	31,0	5,2	36,1	33,0
Max	92050	66020	65,2	65,2	10,9	5,4	34,8
Kr.Pf.	92050	66020	65,2	65,2	10,9	5,4	34,8
Ohne on-spot Mapping							
Min	97050	8020	31,0	67,0	5,2	36,1	33,0
Max	92050	66020	65,2	65,2	10,9	5,4	34,8
Kr.Pf.	92050	66020	65,2	65,2	10,9	5,4	34,8

Die Entscheidung, welches Mapping für den vorgegebenen Datenflussgraphen das Beste ist, kann man der minimalen Ausführungszeit und der Gleichmäßigkeit der Prozessorenauslastung entsprechend treffen. Die gesamte Wartezeit der Knoten $T_{wait\ \sum}$ ist für das MDSP-Rechnersystem kein Kriterium des optimalen Mappings, weil die Knoten, die auf dem Master Prozessor ausgeführt werden müssen, können nur sequentiell ausgeführt werden und deshalb $T_{wait\ \sum}$ gegen die sinkende T_a erhöhen.

Als das beste Mapping wurde on-spot Mapping mit der Strategie „Kritischer Pfad“ ausgewählt und ist mit den verschiedenen Farben der Blöcke im Simulink-Modell gemäß den Prozessoren (Tabelle A.3), die für die Ausführung von diesen benutzt werden, in Abb. A.2 dargestellt.

Tabelle A.3

Die Bedeutungen der Farben der Blöcke des Simulink-Modells (Abb. A.2)

Farbe des Blocks	**Prozessor für die Ausführung**
Blau	Master
Orange	Slave 0
Rosa	Slave 1
Weiß	Slave 2
Gelb	Slave 3

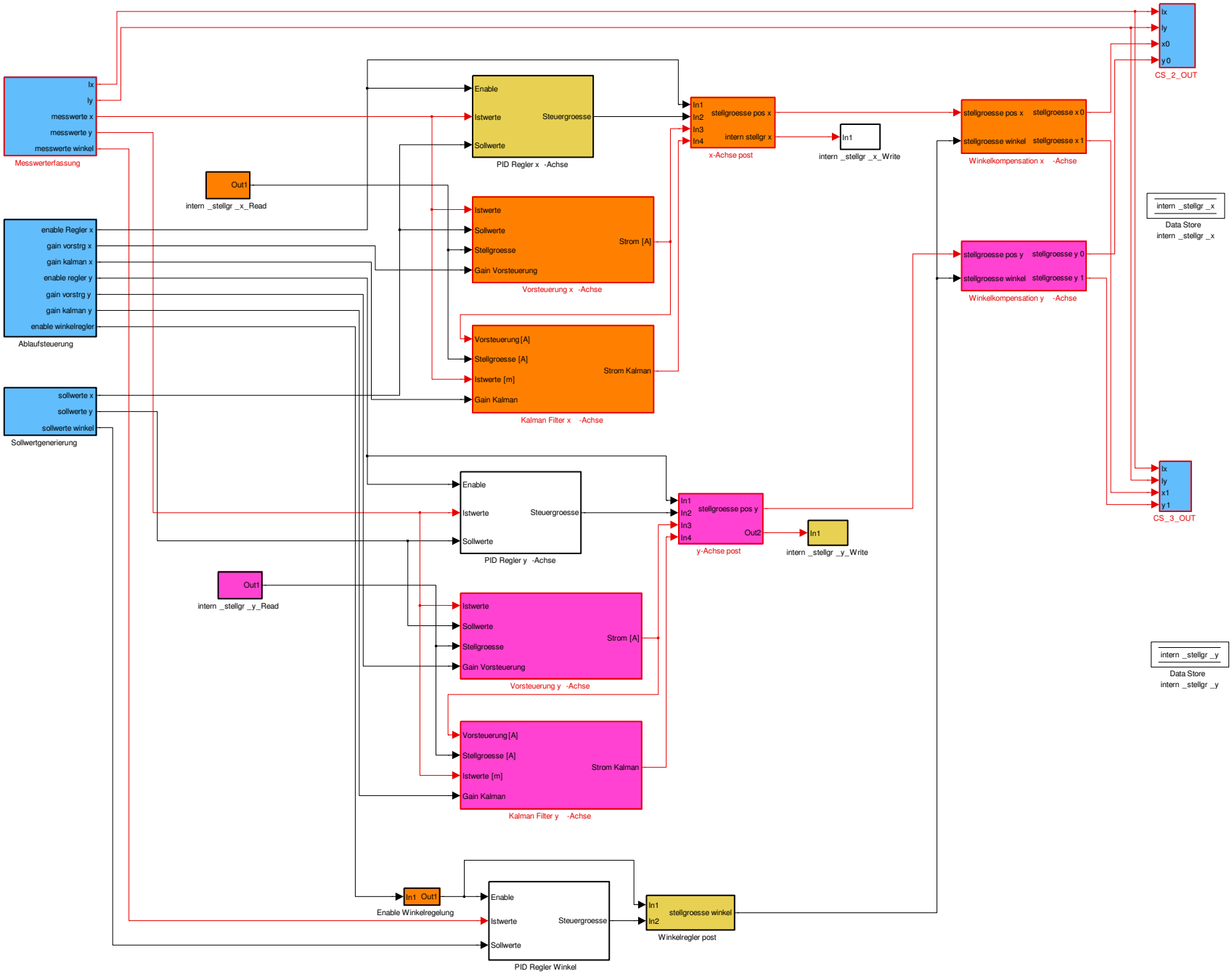


Abb. A.2. Partitioniertes komplexes Reglersystem (Simulink)

Literaturverzeichnis

[1] Bolz, Jan: *Konzeption und Realisierung eines Targets für ein eingebettetes Mehrprozessorsystem*. Technische Univeristät Ilmenau, Diplomarbeit, 2009

[2] Pree, Wolfgang: *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt, 1997

[3] Fayad, Mohamed al; Schmidt, Douglas C.; Johnson, Ralph E.: *Building Application Frameworks*. John Wiley & Sons, Inc., 1999

[4] Müller, Marcus: *Transformation von modellbasierten Systembeschreibungen. Entwurf und Implementation eines Java-Frameworks*. Technische Univeristät Ilmenau, Diplomarbeit, 2004

[5] Ladygin, Igor I.; Kalinina, Galina A.: *Laborarbeiten im Fach „Rechnersysteme“*. Moskauer Energetisches Institut, 1999 *(in russische Sprache)*

[6] Woewodin Wiktor W.; Woewodin Wladimir W.: *Parallel Computing*. BHW-Peterburg, 2004

[7] Eclipse.org home: *http://www.eclipse.org*. Online Ressource, Abruf: 15.01.2010

[8] Wikipedia. Die freie Enzyklopädie: Parser (*http://de.wikipedia.org/wiki/Parser*). Online Ressource, Abruf: 20.02.2010

[9] JavaCC: Documentation Index (*https://javacc.dev.java.net/doc/docindex.html)*. Online Ressource, Abruf: 11.12.2009

[10] The Mathworks: Model File Contents
http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/f22-7548.html. Online Ressource, Abruf: 10.01.2010

[11] The Mathworks: Block Reference
http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/f4-4889.html. Online Ressource, Abruf: 10.01.2010

Danksagung

Diese Danksagung ist eine exzellente Möglickteit mich bei allen Dozenten, Familienmitgliedern und Freunden, die mich im Laufe meines Masterstudiums tatkräftig unterstützt haben, zu bedanken.

Meine Familie hat in meinem Leben einen hohen Stellenwert und aus diesem Grund möchte ich mich in erster Linie bei ihr dafür bedanken, dass sie mich während des Austauschprogramms zwischen dem Moskauer Energetischen Institut und der Technischen Universität Ilmenau fortwährend 4 Jahre unterstützt hat.

Dieses Ausstauschprogramm wurde nur durch die ständige Koordinierung zwischen Frau Ekaterina Dolgatscheva, Herrn Prof. Wolfgang Fengler, Frau Gudrun Matthies**,** Herrn Dipl.-Ing. Klaus-Dieter Fritz und Herrn Dr.-Ing. Alexander Fleischer möglich und bietet den russischen Studenten eine einzigartige Möglichkeit einen Doppelabschluss zu erreichen und damit bessere Chancen für die zukünftige Karriere.

In einer Masterarbeit ist der Forschungsbereich ein wichtiger Bestandteil und deshalb bedanke ich mich bei Dipl.-Inf. Marcus Müller sowie dem gesamten Fachgebiet Rechnerarchitektur für die hervorragende Betreuung und das interessante Thema meiner Masterarbeit.

Für die intensive Korrektur der Grammatik meiner Masterarbeit und dem dadurch entstandenen hohen Zeitaufwand neben seinem Studium, bedanke ich mich bei Robin Zeh (Mechatronik BA M-09).

Dankbar bin ich außerdem Kevin Kokonowski (Wirtschaftsinformatik BA M-06) für seine Hilfestellung bei der deutschen Rechtschreibung und Grammatik.

Sehr verbunden bin ich Andrey Dratz (Staatliche Universität Petrozawodsk, Russland) für seine Unterstützung bei der technischen Realisierung von Programmen in Java.

Zum Abschluss möchte ich ein Dankeswort an Irina Guschtschina (Moskauer Energetisches Institut, Russland) richten für die Inspiration zum Schreiben, die sie mir gegeben hat.

Erklärung

Hiermit erkläre ich, Andrey Kondrat, dass ich diese Masterarbeit selbständig und nur unter Verwendung der angegebenen Quellen anfertigte.

Ilmenau, den 30.04.2010 Andrey Kondrat

Thesen

1. Das Problem der optimalen Abbildung von einer Menge von parallelisierbaren Funktionen auf eine Menge von Berechnungsressourcen ist NP-vollständig und wird normalerweise heuristisch durch die Kombination von Graphenanalyseverfahren gelöst.
2. Ein Datenflussgraph ist eine universelle Darstellung, in die jedes beliebige Eingangsdatenformat mit dem jeweiligen Parser umgewandelt werden kann und die von der Architektur des Zielrechnersystems unabhängig ist.
3. Die Partitionierung des Datenflussgraphen auf das Zielrechnersystem erfolgt durch die Transformierung des Eingangsmodells, das die Beschreibung des Datenflussgraphen und der Zielplattform enthält, ins Zielmodell, das aus dem Eingangsmodell mit dem Mapping erweitert ist. Die Partitionierung wird der vom Nutzer der entwickelten Applikation ausgewählten Ausführungsstrategie entsprechend durchgeführt.
4. Die Anwendung von Frameworks in der entwickelten Applikation eröffnet einen Weg, auf dem es möglich ist, das Wesen erfolgreicher Architekturen, Entwurfmuster, Komponenten und Programmiermechanismen wiederverwendbar zu machen.
5. Der Prozess der Partitionierung des Datenflussgraphen modelliert das Verhalten des realen Rechnersystems, als ob auf diesem der vorgegebene Datenflussgraph als Task ausgeführt wird, und berücksichtigt dabei die Hierarchie von Rechner- und Kommunikationsressourcen (Prozessoren und Busse).
6. Durch die Verwendung der qualitativen und quantitativen Datenflussgraphenanalyseverfahren werden solche wichtigen Parameter wie die minimal mögliche Ausführungszeit des Graphen auf dem Rechnersystem mit unbegrenzten Ressourcen (T_{min}) und der maximale Speedup berechnet, damit man entweder die minimale Ausführungszeit des Graphen auf dem vorgegebenen Rechnersystem bewerten oder das optimale Rechnersystem, auf dem der Graph in T_{min} ausgeführt werden kann, finden kann.

Ilmenau, den 30.04.2010 Andrey Kondrat